MASTER EN SCIENCES INFORMATIQUES

Outil informatique annexe à une méthode d'initiation aux raisonnements de la programmation

Vandenbroucke, Anne

Award date:
1991

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

OUTIL INFORMATIQUE ANNEXE A UNE METHODE D'INITIATION AUX RAISONNEMENTS DE LA PROGRAMMATION

par

Anne de Baenst-Vandenbroucke

Mémoire présenté en vue de
l'obtention du diplôme de
LICENCIE ET MAITRE EN INFORMATIQUE
(cycle de deux ans)

PROMOTEUR :
Monsieur Roland Lesuisse

Année Académique 1990 - 1991

RESUME

Ce Mémoire concerne le développement d'un outil annexe à la *Méthode d'Initiation aux Raisonnements de la Programmation* mise au point par R. Lesuisse et A. Borsu dans le cadre de leur cours d'Introduction à l'Informatique permettant d'intégrer la méthode dans un contexte d'Enseignement Assisté par Ordinateur (EAO).

L'outil développé est destiné à illustrer sur ordinateur comment un ordinateur procède pour exécuter un programme ou algorithme associé à un problème en prenant le Modèle Général de l'Ordinateur de Lesuisse et Borsu comme cadre de référence. Le langage de programmation reconnu par l'outil se présente comme un mini langage limité aux modes de raisonnements fondamentaux. La version opérationnelle de l'outil utilise une traduction de convention Pascal de ce mini langage.

Le mémoire a été rédigé en mettant l'accent sur l'aspect pédagogique des concepts mis en jeu.

ABSTRACT

This work concerns the development of a tool annex to the *"Méthode d'Initiation aux Raisonnements de la Programmation"* established by R. Lesuisse and A. Borsu in the context of their lessons in the Introduction of Programming Science and situating the method in an environment of Computed Aided Learning (CAL).

The developped tool is intended to show on a computer how a computer manages when executing a program or algorithm associated with a problem taking the "Modèle Général de l'Ordinateur" of Lesuisse and Borsu as working scheme. The programming language used for the tool is a mini language resumed to the fundamental ways of reasonning. The operating version of the tool works with a Pascal translation of the mini language.

The work has been written by giving attention on the pedagogical aspect of the involved notions.

Entre ce que l'on peut et ce que l'on veut,
serpente l'étroit chemin de notre réalisme.

G. Goldman, *Quel métier pour votre enfant?*, Laffont, Paris, 1990

REMERCIEMENTS

Je remercie tout particulièrement Monsieur R. Lesuisse de bien avoir voulu accepter la direction de ce Mémoire et d'avoir pris sur son temps pour lire au plus vite un manuscrit présenté dans des délais non prévus.

Je remercie également Madame A. Borsu qui a toujours répondu avec gentillesse à toute mes demandes lors de la phase de développement du mémoire.

Je remercie aussi Monsieur J. Fichefet pour sa grande compréhension au cours de la rédaction de ce mémoire.

Je remercie encore Monsieur F. de Azevedo et Madame A.M. Guyot pour leurs encouragements durant ces derniers mois ainsi que tous ceux qui, d'une façon ou d'une autre, m'ont aidée à aller jusqu'au bout.

Je remercie surtout Monsieur P. de Baenst d'avoir supporté avec philosophie les longues heures passées en solitaire et les sautes d'humeur d'une épouse absorbée par l'entreprise pleine d'embûches d'un mémoire à un âge dit "respectable" ainsi que pour son rôle de "candide" terriblement critique mais fort efficace.

Je remercie enfin Messieurs D. et B. de Baenst d'avoir accepté avec philosophie d'être un peu négligé par leur mère au profit d'un ordinateur bien accaparant.

PLAN GENERAL

TABLE DES MATIERES

TABLE DES MATIERES

INTRODUCTION

Ce Mémoire traite du développement d'un outil informatique annexe à la *Méthode d'Initiation aux Raisonnements de la Programmation* mise au point par R. Lesuisse et A. Borsu dans le cadre de leur cours d'Introduction à l'Informatique [A/LE87].

L'outil informatique concerné est un outil destiné à illustrer sur *ordinateur* comment un *ordinateur* procède pour exécuter un programme ou algorithme associé à un problème, et cela, en prenant comme cadre de référence le **Modèle Général de l'Ordinateur** mis au point par Lesuisse et Borsu pour leur méthode d'initiation.

Quoi de plus naturel pour un mémoire en Informatique que de réaliser un projet informatique et le présent mémoire ne déroge pas à cette constatation. Mais, dans ce cas, il s'ajoute une dimension supplémentaire: le projet informatique porte sur l'apprentissage de l'informatique. L'ordinateur est non seulement l'instrument de travail, mais il est aussi le sujet du travail. L'abstrait, le concret, tout se mêle, tout se mélange. On peut vite se rendre compte que la réalisation d'un projet tel que celui qui nous concerne, met en jeu plusieurs domaines de la science de la programmation tels que facteurs humains dans l'enseignement de la programmation, méthodologie du développement de logiciels, enseignement assisté par ordinateur (EAO), algorithmique et langage de programmation, théorie des langages de programmation, étude approfondie du langage Pascal, réalisation de logiciels interactifs.... et il n'a pas été facile de décider comment

documenter le travail. Nous avons finalement trouvé intéressant de le faire en adoptant une approche pédagogique qui pourra, nous l'espérons, faire ressortir la bivalence qui existe entre les notions traitées dans le projet et les notions utilisées pour le réaliser. N'est-ce-pas, en fin de compte, la meilleure façon pour apprendre à construire un logiciel d'enseignement de la programmation susceptible de s'intégrer dans un cadre d'enseignement assisté par ordinateur?

Pratiquement, dans le premier chapitre, nous exposons tous les concepts de la *Méthode d'Initiation aux Raisonnements de la Programmation de Lesuisse et Borsu* [A/LE87] susceptibles, d'une part, de mieux comprendre leur **Modèle Général d'un Ordinateur** et son utilisation et, d'autre part, d'introduire les notions permettant de montrer comment passer de l'exécution d'un problème simple à la réalisation d'un projet plus vaste.

Dans le deuxième chapitre, nous arrivons alors tout naturellement à la présentation générale des différentes étapes relatives à la conception de l'outil informatique souhaité par Lesuisse et Borsu. Comme déjà mentionné, l'accent est mis sur la construction de développement de l'outil comme généralisation des notions contenues dans le premier chapitre plutôt que sur la présentation technique des détails de réalisation. Nous nous sommes plus particulièrement attachée à la présentation du projet d'un outil informatique, à la définition d'un cadre général, à l'analyse fonctionnelle ainsi qu'à la présentation de l'architecture logique retenue.

Dans le troisième chapitre, nous faisons une étude détaillée du mini langage qui a été retenu pour l'outil. Ici encore, l'accent est mis sur la construction de ce langage en partant des notions abstraites pour arriver en fin de compte à une définition concrète susceptible d'être intégrée dans l'outil. Cette approche est guidée par le désir de construire un outil capable d'exécuter des programmes écrits dans divers langage de programmation. Dans ce chapitre nous donnons deux

syntaxes concrètes, une syntaxe de convention Pascal, appelée *DEMO PASCAL*, ainsi qu'une syntaxe pour un langage formel aussi proche que possible du langage formel utilisé dans la méthode d'initiation de Lesuisse et Borsu. Dans la version actuelle, seul le *DEMO PASCAL* est déjà intégré.

Dans le quatrième chapitre, nous donnons un guide de l'utilisateur qui devrait permettre à n'importe quel utilisateur non initié à la méthode de Lesuisse et Borsu d'au moins comprendre le pourquoi et le comment de l'outil développé. Ce guide permet de montrer le produit réalisé.

Dans la conclusion, nous terminons par une courte évaluation de fin de travail sur l'outil produit à la fois par rapport à lui-même et par rapport à sa contribution dans le domaine de l'apprentissage de l'Informatique.

Après une bibliographie reprenant les différents travaux que nous avons consultés pour la réalisation du mémoire, nous donnons en annexe une liste de quelques petits programmes permettant d'illustrer la version actuelle de l'outil, le codage de l'outil ainsi que la définition du langage du *DEMO PASCAL* suivant les diagrammes syntaxiques.

Maître d'arithmétique (*Gossuin de Metz, De imagine mundi, milieu du XII^e^ siècle*)

Chapitre 1

PRESENTATION DU MODELE GENERAL D'UN ORDINATEUR DE LESUISSE ET BORSU

1.1 Introduction.

Dans le cadre de leur cours d'*Introduction à l'Informatique* délivré aux étudiants de Première Candidature en Sciences Economiques, Politiques et Sociales des Facultés Universitaires Notre-Dame de la Paix de Namur, Lesuisse et Borsu ont mis au point et expérimenté depuis quelques années une méthode originale d'*Initiation aux Raisonnements de la Programmation* fondée sur quelques modèles classiques du comportement cognitif de l'*Homme*, élaborés par des disciplines telles que la psychologie, la linguistique et même, la philosophie. Leur méthode est exposée en détail dans un ouvrage dont la première édition a paru en 1987 aux Presses Universitaires de Namur sous le titre: *Initiation aux Raisonnements de la Programmation* [A/LE87].

L'idée de base de la méthode consiste à privilégier l'acquisition progressive de concepts fondamentaux et de modes de raisonnement de base, plutôt que l'acquisition d'un langage de programmation donné, en l'occurrence le *Pascal*, bien que cet aspect ne soit pas pour autant négligé.

Pour Lesuisse et Borsu, cette acquisition progressive de concepts fondamentaux et de modes de raisonnement de base est consolidée si l'étudiant aborde son étude en ayant quelques idées fondamentales sur la *structure* et le *mode de*

fonctionnement de l'outil de base, à savoir l'*ordinateur*. C'est pourquoi, dans leur ouvrage, ils proposent un **Modèle Général d'un Ordinateur** permettant aux étudiants de réaliser comment l'ordinateur travaille. Leur modèle est volontairement très schématique et n'a pas l'ambition de représenter dans tous les détails, un vrai ordinateur. Comme ils l'écrivent dans leur introduction(1), "... Cette description se limite aux éléments qu'il est indispensable de connaître pour programmer cette machine. ...". Leur modèle se rattache au courant d'idées actuel qui consiste à dire que dans une démarche d'apprentissage performante, l'homme doit être guidé par le **POURQUOI** plutôt que par le **COMMENT**(2).

Dans ce premier chapitre, nous reprennons les grandes lignes de ce **Modèle Général d'un Ordinateur** de Lesuisse et Borsu. En nous référant à leur travail [A/LE87], nous commençons par situer la machine ordinateur par rapport à l'exécutant humain. Nous énumérons ensuite les éléments constituants de l'ordinateur pour en arriver à la présentation du modèle. Pour terminer, nous exposons en détails une résolution d'un problème simple dans le cadre du modèle.

1.2 La machine Ordinateur.

Dans une première approximation, un *ordinateur* peut être assimilé à une *machine* destinée à *exécuter*, en lieu et place d'un exécutant humain, une *marche à suivre* permettant la résolution d'un problème. La marche à suivre de la résolution, appelée *algorithme*, est un *texte* comprenant la description, dans un certain *langage*, d'une *suite d'opérations* portant sur des *objets* et devant être exécutées dans un certain *ordre chronologique*.

- Les *opérations* doivent être *primitives*, c-à-d telles qu'un exécutant donné soit capable de les accomplir sans aide

---

(1) Cf. [A/LE87], p. 14.

(2) La section B de la Bibliographie fait l'inventaire des ouvrages et des articles que nous avons eu l'occasion de consulter parmi l'abondante littérature se rattachant à ce courant d'idées et qui nous ont permis de mieux en percevoir la portée.

extérieure; cette capacité d'accomplissement peut être soit innée soit acquise par apprentissage.

- Les *objets* manipulés sont de type *informations*; une information est un ensemble structuré de signes ou de symboles ayant un *sens* et une *forme* bien précis[3].
- L'*ordre chronologique* d'exécution de la suite des opérations s'identifie, en général, à l'ordre d'apparition de ces opérations dans le texte de l'algorithme. Les opérations sont alors exécutées, dans le temps, les unes après les autres une et une seule fois. Il se peut qu'une opération ou sous-suite d'opérations soit *conditionnée* avec, comme conséquence, de ne pas être exécutée dans certains cas. Il se peut également qu'une opération ou une sous-suite d'opérations soit *à répéter* , ce qui consiste à l'exécuter consécutivement un nombre déterminé de fois (éventuellement même aucune fois).

On notera également que, tout comme un exécutant humain, l'*ordinateur* est doté des caractéristiques suivantes:

- Il est capable de *prendre connaissance* d'un algorithme et de *comprendre le langage* dans lequel l'algorithme est écrit;
- Lorsqu'il a pris connaissance d'un algorithme, il est capable de le *mémoriser* le temps de son exécution[4];
- Il est capable d'enregistrer des informations *émanant* de l'utilisateur;
- Il est capable de *transmettre* à l' utilisateur le résultat de l'exécution d'un algorithme;
- Il est capable de *conserver à long terme* un algorithme ou des informations pour une utilisation ultérieure[5];

---

(3) Dans [A/LE87], Lesuisse et Borsu retiennent comme forme, soit celle d'un nombre, entier ou réel, soit celle d'une suite de caractères.

(4) On peut ajouter qu'il est également capable de reproduire immédiatement l'exécution *autant de fois* qu'on le souhaite. Comme l'indiquent Shneiderman et Mayer [B/SH79], la capacité d'effectuer la première exécution relève du domaine de la mémoire de travail (working memory) tandis que la reproduction immédiate de l'opération relève cette fois du domaine de la mémoire à court terme (short-term memory).

(5) Si, comme dans le note précédente, on s'en réfère au travail de Shneiderman et Mayer [B/SH79], cette caractéristique relève du domaine de la mémoire à long terme (long-term memory).

enfin
- sa capacité ne se limite pas à exécuter un seul algorithme; il est évidemment capable de traiter une *multitude* d'algorithmes différents.

1.3 Les éléments constituants de l'Ordinateur.

Pratiquement, un *ordinateur* est constitué d'une part, du *hardware* et d'autre part, du *software*.

Le *hardware* est composé de tous les *éléments matériels* permettant de prendre connaissance de la marche à suivre d'un programme ou d'informations, de la(les) mémoriser, d'exécuter des opérations, de transmettre des résultats à un utilisateur et également, de conserver un programme ou des informations à long terme. Parmi tous ces éléments matériels, il est habituel de distinguer:
- d'une part, l'*unité centrale* constituée des éléments capables de mémoriser à court terme un programme et d'en exécuter les opérations,
- d'autre part, les *unités périphériques* regroupant tous les autres éléments; parmi celles-ci, on distingue:
  - les *unités de communication* regroupant les éléments permettant de prendre connaissance d'un programme ou d'informations ainsi que ceux susceptibles de transmettre un résultat à l'utilisateur;
  - les *unités capables de conserver à long terme* un programme ou de l'information.

Ces éléments matériels sont généralement assemblés selon le schéma représenté à la figure *Fig. 1.1*.

Le *software* consiste en des *éléments logiciels* réalisés au moyen de programmes; ce sont ces éléments logiciels qui permettent à un ordinateur de *comprendre le langage* dans lequel un programme est écrit.

Dans ce mémoire, nous nous intéresserons plus particulièrement à la **modélisation** de l'*unité centrale* qui est l'élément matériel essentiel de l'ordinateur, ainsi que des

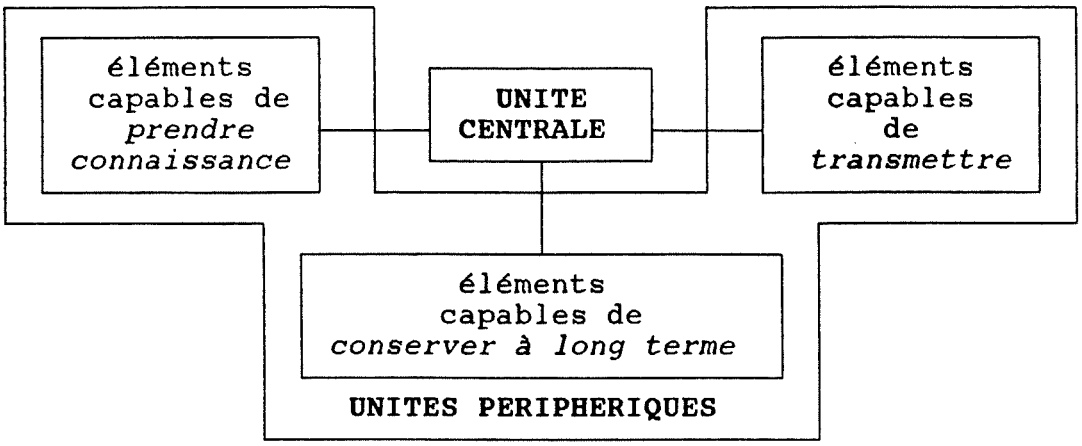


Fig. 1.1 Schéma général des éléments matériels d'un ordinateur

unités de communication permettant à un utilisateur de communiquer avec cet ordinateur(6).

1.4 Modèle Général d'un Ordinateur.

1.4.1 L'Unité Centrale.

L'*unité centrale* d'un ordinateur est formée de deux éléments complémentaires, la *mémoire centrale* et le *processeur central* (*Fig. 1.2*).

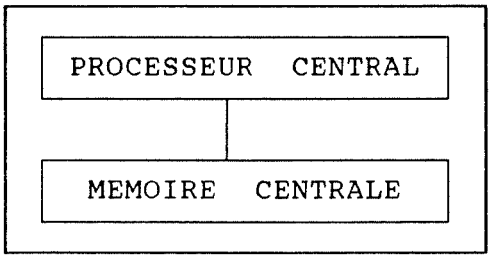


Fig. 1.2 Schéma général de l'unité centrale

La *mémoire centrale* a, comme son nom l'indique, une fonction de mémorisation. Elle sert à mémoriser toutes les informations nécessaires pour que l'ordinateur puisse exécuter

---

(6) On trouvera plus de détails sur les autres éléments constituants dans [A/LE87] aux pages 49 à 60.

un programme. Le *processeur central* , pour sa part, remplit une fonction de gestion et d'exécution du travail.

1.4.1.1 *La Mémoire Centrale.*

La *mémoire centrale* sert:

- d'une part, à stocker les informations émanant de l'utilisateur ainsi que les résultats obtenus en cours d'exécution d'un programme,
- d'autre part, à mémoriser les opérations à exécuter qui sont elles-même transmises au moyen du texte de l'algorithme.

La partie de la *mémoire centrale* servant à stocker les informations et résultats peut être modélisée (*Fig. 1.3*) sous la forme d'un grand *casier* comportant un certain nombre de *cases individuellement identifiables* par une étiquette ou *nom* de la case. Lesuisse et Borsu font l'hypothèse que, dans chaque case, on peut ranger soit un nombre (entier ou réel), soit un caractère. Ce nombre ou ce caractère est le contenu de la case et est appelé *valeur* de la case. Chaque case a donc un *nom* et une *valeur* (*Fig. 1.4*).

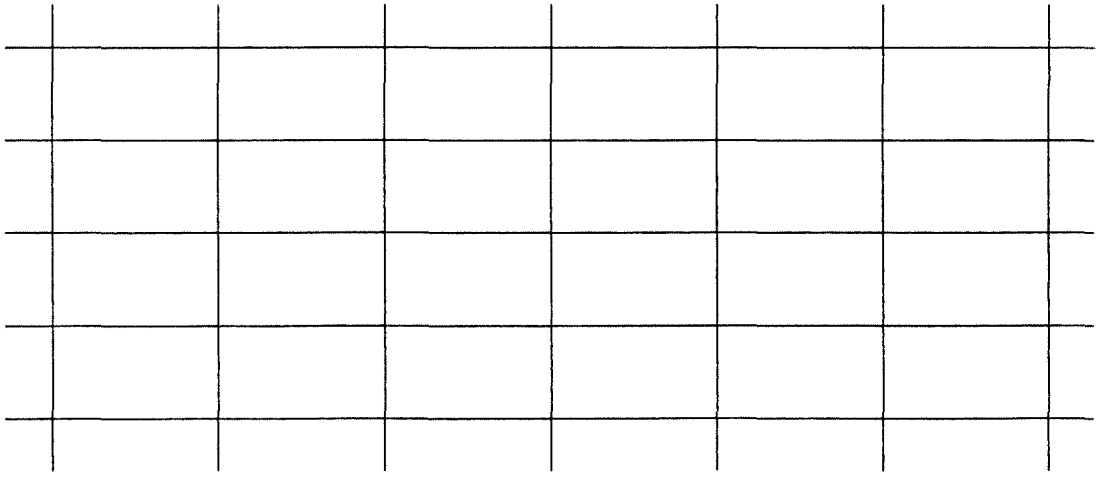


Fig. 1.3 Schéma de la partie "informations" de la mémoire centrale

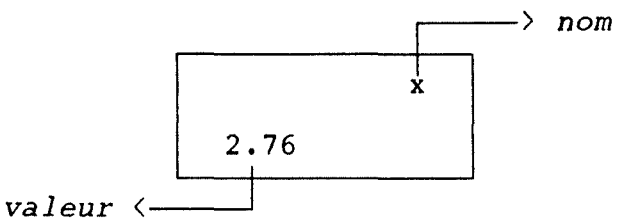


Fig. 1.4 Case de nom "x" et de valeur "2.76"

Pour des raisons pédagogiques, Lesuisse et Borsu proposent de modéliser la partie de la *mémoire centrale* servant à mémoriser les opérations à exécuter en reproduisant, dans un cadre, le texte de l'algorithme en supposant implicitement au plus une opération primitive par ligne de texte.

Les deux parties de la *mémoire centrale* sont rassemblées sur un seul et même schéma et forment le *schéma général de la mémoire centrale* (*Fig. 1.5*).

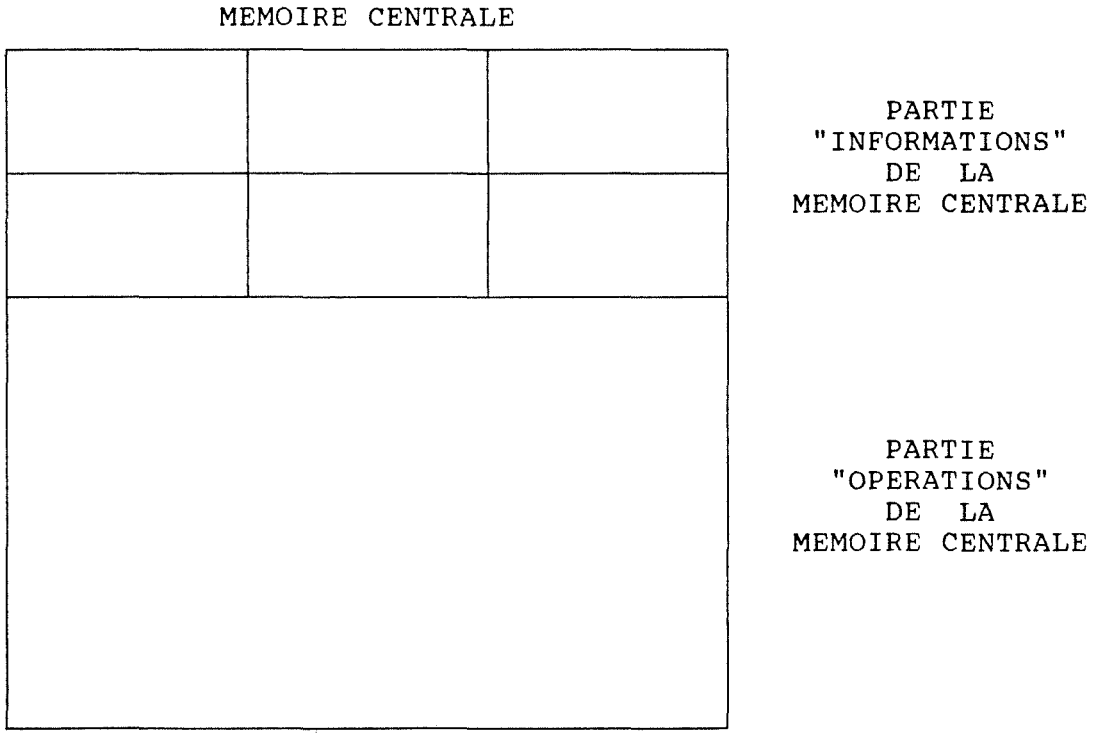


Fig. 1.5 Schéma général de la Mémoire Centrale

Le nombre de cases contenues dans les deux parties de la *mémoire centrale* dépend, en principe, des capacités de l'ordinateur. Comme le mentionnent Lesuisse et Borsu(7), "... Typiquement ce nombre peut varier de quelques milliers à quelques millions ...". Il est clair qu'il ne serait pas possible de visualiser sur un seul schéma la mémoire centrale en son entier. Ce qu'on souhaite au moins représenter, c'est la partie relative au programme en cours d'exécution.

La *mémoire centrale* est limitée en *étendue* par la capacité de l'ordinateur. On peut remarquer que cette limitation est également présente dans le cas de la mémoire humaine. Cependant, contrairement à la mémoire humaine, la *mémoire centrale* de l'ordinateur est également limitée en *durée*. Elle est une mémoire exclusivement à court terme. En effet, une fois que l'ordinateur n'est plus sous tension, tout s'efface de la mémoire. Pour la mémorisation à long terme, l'ordinateur dispose d'autres éléments matériels faisant partie des unités périphériques (8).

1.4.1.2 *Le Processeur Central.*

Le *processeur central* remplit deux fonctions distinctes:
- d'une part, il *gère l'ordre chronologique* d'exécution des opérations primitives figurant dans le texte du programme en cours d'exécution,
- d'autre part, il *exécute* ces opérations.

Le *processeur central* remplit ces deux fonctions grâce à divers *circuits imprimés* qu'il est évidemment difficile de représenter sur un schéma. Pour des raisons de commodité, dans le modèle de Lesuisse et Borsu, les circuits qui gèrent l'ordre d'exécution des opérations sont représentés au moyen d'un signet "->" figurant dans la *mémoire centrale* en face de l'opération à exécuter ou en cours d'exécution (*Fig. 1.6*).

---

(7) Cf. [A/LE87], p. 52.

(8) Cf. [A/LE87], section 2.3.3, pages 55 à 59.

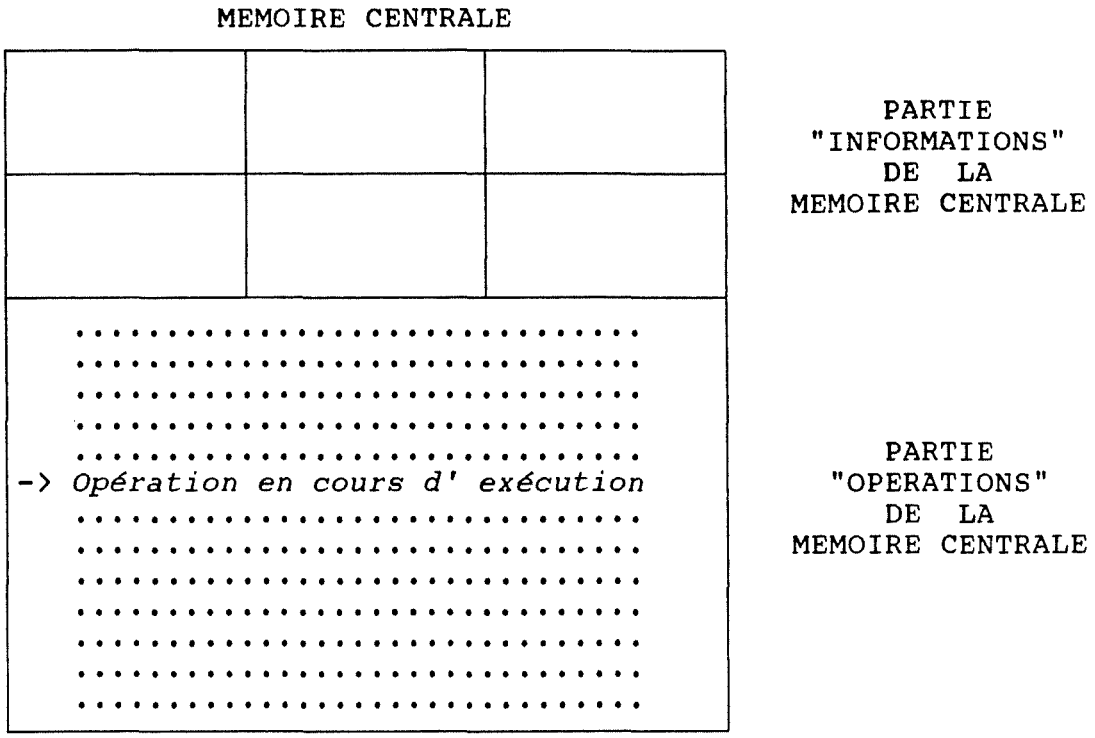


Fig. 1.6 Représentation par un signet des circuits qui gèrent l'ordre d'exécution

En ce qui concerne les circuits qui gèrent l'*exécution* proprement dite *des opérations primitives* figurant dans le texte du programme, il est difficile d'en donner une représentation explicite. Il est cependant possible d'observer le résultat de l'exécution au vu des modifications qui s'opèrent au niveau de la *partie "informations" de la mémoire centrale*.

On observera ces modifications sur un exemple d'exécution de programme. Avant cela, il reste encore à représenter les modes de communications entre l'*utilisateur humain* et l'*unité centrale*.

1.4.2 Les Unités de Communications.

Les *unités de communications* ont pour fonction essentielle d'assurer l'échange entre l'*ordinateur* et son *utilisateur humain*. Cet échange s'applique sur le programme, sur les informations qu'il nécessite ainsi que sur les résultats qui en découlent.

Pour l'échange dans le sens *homme-ordinateur*, le dispositif utilisé est le *clavier*. Il est mis à contribution chaque fois qu'il y a demande d'informations.

Pour l'échange dans le sens *ordinateur-homme*, les deux sortes de dispositifs sont l'écran et l'imprimante. Lesuisse et Borsu retiennent uniquement l'*écran*. Celui-ci sert d'une part, d'écho à la prise d'informations à partir du clavier et d'autre part, à l'affichage des résultats obtenus lors de l'exécution d'un programme.

Afin de visualiser tous les échanges entre l'utilisateur et l'ordinateur, ces deux auteurs proposent une représentation des dispositifs de communication sous la forme d'un *ensemble* ECRAN/CLAVIER (*Fig. 1.7*)(9). C'est évidemment cet ensemble qui est le seul apparent pour l'utilisateur humain.

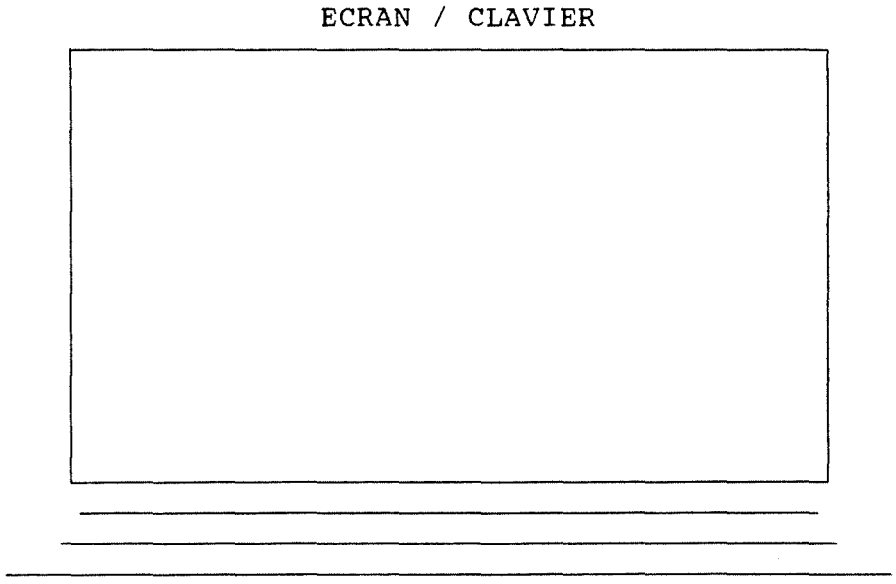


Fig. 1.7 Schéma des éléments de communications

Nous disposons à présent des éléments essentiels du **Modèle Général d'un Ordinateur** permettant d'avoir une meilleure

(9) Ce schéma est utilisé pour la première fois à la page 72 de [A/LE87].

compréhension de la *structure* et du *mode de fonctionnement* de l'*ordinateur*.

1.5 Résolution d'un problème dans le cadre du Modèle.

1.5.1 Modalités.

Dans cette section, nous montrons sur un exemple la *résolution* complète *d'un problème* en utilisant les différents éléments du **Modèle Général d'un Ordinateur**. Nous prendrons comme exemple le "Premier Programme" étudié dans le chapitre 3 de l'ouvrage de Lesuisse et Borsu[10].

Afin de rédiger le texte de l'algorithme, il faut au départ faire le choix d'un langage de représentation d'algorithme appelé *langage de programmation*. Ce choix ne modifie en rien l'algorithme proprement dit. Le langage sert en fait à permettre le dialogue entre l'utilisateur et l'ordinateur. Il est compris par l'ordinateur grâce aux *éléments logiciels*. Pour bien montrer que le **Modèle** n'est pas lié à un langage particulier, la mise en oeuvre du "Premier Programme" sera effectuée avec le *langage formel* adopté dans l'ouvrage de référence[11] ainsi qu'en *langage Pascal*.

Pour résoudre un problème, il faut disposer *d'un énoncé* clair *du problème*. Ensuite, il s'agit de *construire* l'*algorithme* et le *rédiger dans un langage de programmation* au choix, ici donc en *langage formel* et en *langage Pascal*. Ces deux étapes achevées, il est alors possible de *procéder à l'exécution* proprement dite du problème.

1.5.2 Enoncé du problème.

Le problème considéré s'énonce ainsi:

"Etant donné

. 2 nombres introduits par l'utilisateur

---

(10) Cf. [A/LE87], Chapitre 3, Pages 63 et suivantes.

(11) Pour plus de détails sur le *langage formel*, on consultera soit [A/LE87], soit le chapitre 3 du présent mémoire.

on demande de construire un programme qui calcule et communique à l'utilisateur
. leur somme."

1.5.3 Construction de l'algorithme.

Dans ce problème, on peut isoler *trois opérations primitives*:

- l'**opération de lecture** qui a pour effet d'affecter à une case une valeur transmise par l'utilisateur (on parle également d'opération d'affectation *externe*),
- l'**opération d'affectation** qui a pour effet d'affecter à une case une valeur trouvée à partir de constantes ou de valeurs déjà rangées dans la mémoire centrale (c'est donc une opération d'affectation *interne*),
- l'**opération d'écriture** ou **d'affichage** qui a pour effet de communiquer à l'utilisateur la valeur associée à une case de la mémoire centrale.

Le vocabulaire utilisé pour formuler ces opérations primitives diffère suivant le langage retenu mais, quelle que soit cette formulation, l'effet de l'opération est identique.

La manière dont les cases de type "informations" de la mémoire centrale sont réservées, dépend également du langage. Ainsi, dans le *langage formel* les cases sont réservées en cours de traitement des opérations tandis qu'en *langage Pascal*, toutes les cases le sont déjà avant le début de l'exécution proprement dite. Cette constatation se manifeste déjà au niveau du texte de l'algorithme mais elle ne modifie en rien le résultat final de l'exécution du problème et relève uniquement de conventions relatives aux *éléments logiciels*.

Pour la représentation de l'algorithme, on reconnait l'intérêt de veiller à la *lisibilité* du texte. Il est ainsi vivement conseillé de ne pas faire figurer plus d'une opération primitive par ligne de texte. Dans le cas du problème "Premier Programme", on retiendra les deux représentations suivantes de l'algorithme:

PREMIER PROGRAMME

Deux Représentations de l'Algorithme

Représentation en langage formel :

```
lire a
lire b
c <- a + b
afficher c
```

avec a, b et c des nombres réels.

Représentation en langage Pascal :

```
program sommation;
var
   a,b,c : real;

begin
   read(a);
   read(b);
   c := a + b;
   write(c);
end.
```

Il reste à voir maintenant comment l'ordinateur travaille pour exécuter l'algorithme.

1.5.4 Avant l'exécution.

Lorsqu'on allume l'ordinateur, on se trouve dans la situation illustrée à la figure *Fig. 1.8*. Le curseur est situé sur l'ECRAN en haut et à gauche prêt à communiquer les informations provenant des échanges d'informations. La MEMOIRE CENTRALE est *vide d'informations*. Il faut cependant faire attention à la signification de l'expression *"vide d'informations"*. En réalité, les cases ont toujours un contenu, mais, au départ, ce contenu est entièrement *aléatoire* et dépourvu de sens. C'est pourquoi, il faut être attentif à définir convenablement dans l'algorithme le contenu des *cases* d'*informations*, tant le *nom* que la *valeur*. Dans le cas contraire, il pourrait se produire des résultats totalement imprévisibles! Les cases *vides d'informations* sont représentées dans le modèle au moyen du sigle "??".

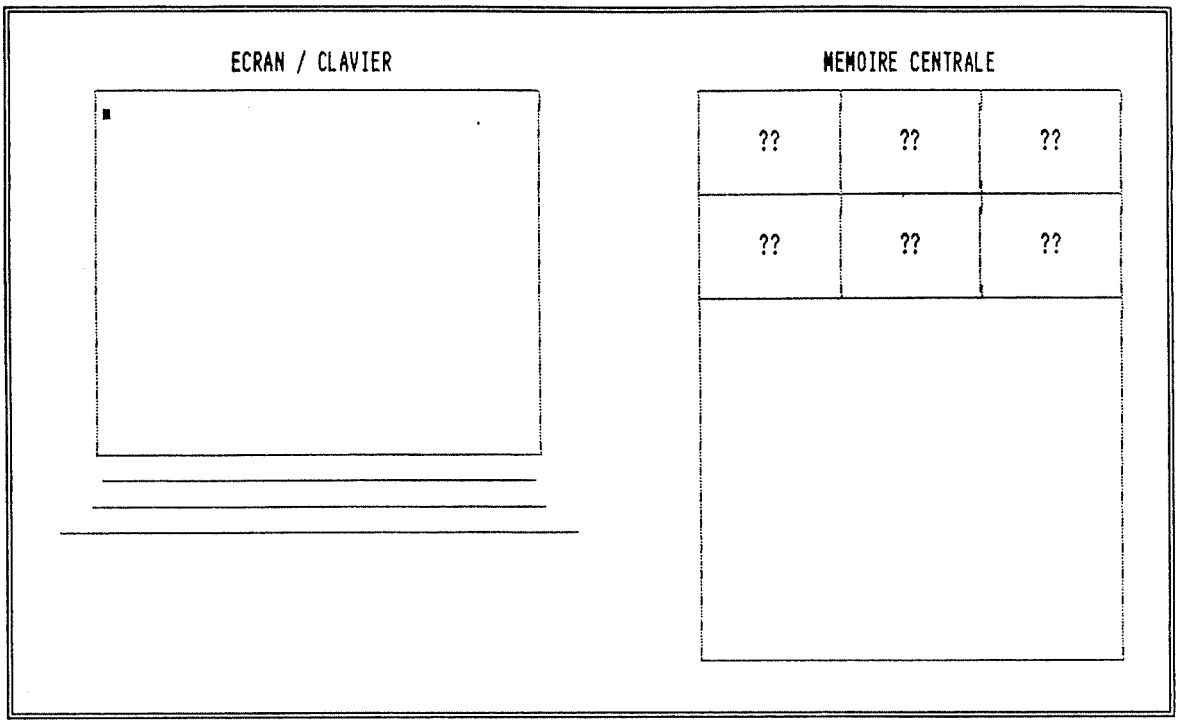


Fig. 1.8 Modèle Général de l'Ordinateur
(Situation à l'allumage de l'ordinateur)

Avant de procéder à l'exécution proprement dite, le texte de l'algorithme est introduit dans la *partie "opérations"* de la MEMOIRE CENTRALE. Suite à cet enregistrement, le **Modèle Général** se présente de la façon suivante (*Fig. 1.9a* et *Fig 1.9b*):

PREMIER PROGRAMME

Deux Situations Avant l'Exécution

Avec le *langage formel* :

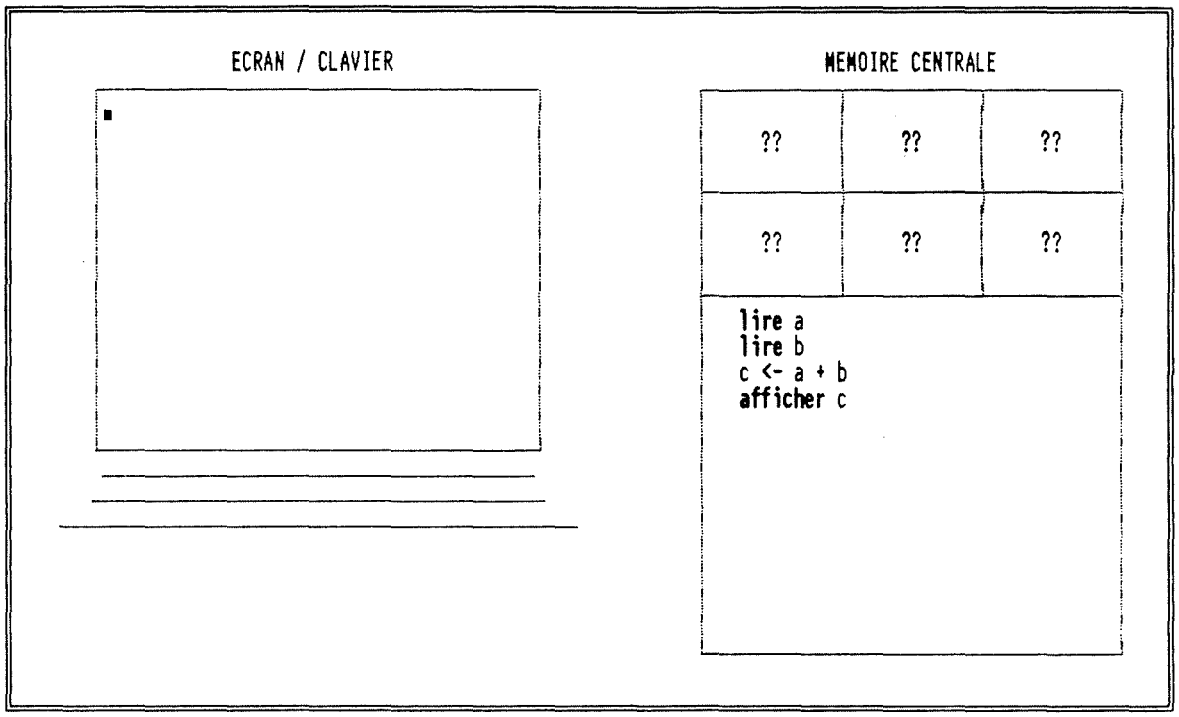


Fig. 1.9a Situation après l'enregistrement du texte de l'algorithme (algorithme en langage formel)

Avec le *langage Pascal* :

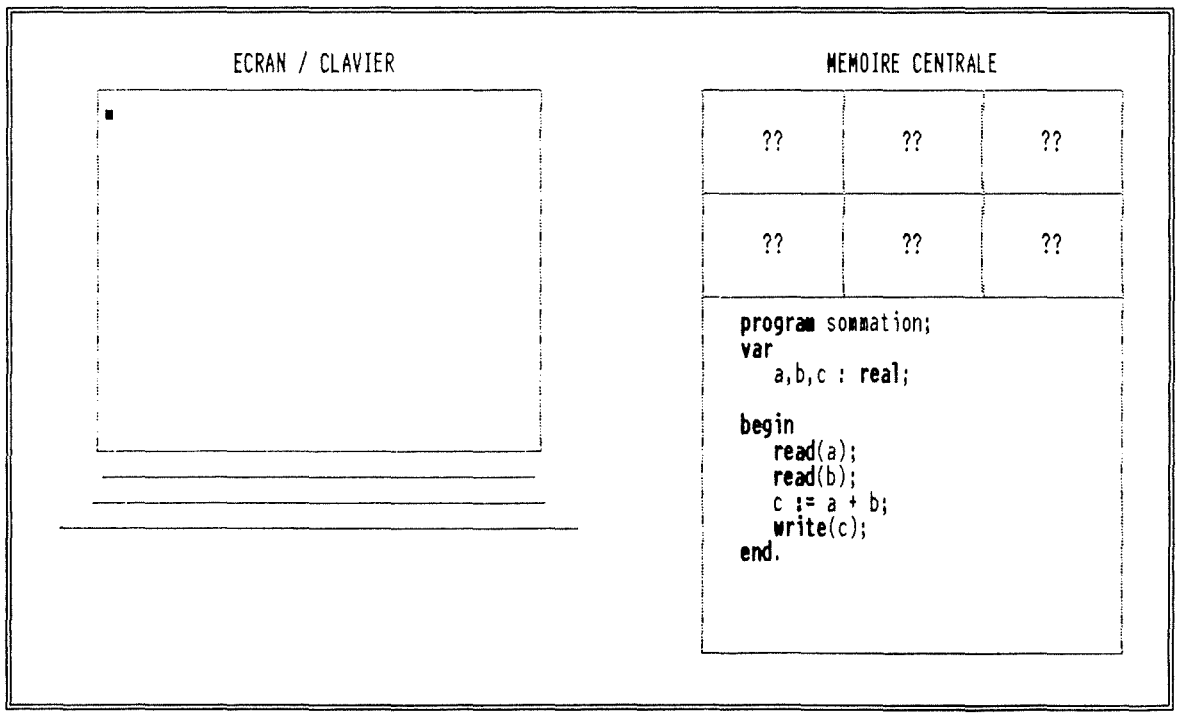


Fig. 1.9b Situation après l'enregistrement du texte de l'algorithme (algorithme en langage Pascal)

Une fois le texte de l'algorithme mémorisé, on peut passer à l'exécution proprement dite.

1.5.5 Exécution.

Voici le scénario de l'exécution du problème pour les deux représentations de l'algorithme (*Fig. 1.10a* à *Fig. 1.10e* pour l'exécution de la représentation en *langage formel* et *Fig. 1.11a* à *Fig. 1.11j* pour l'exécution de la représentation en *langage Pascal*).

Comme décrit précédemment[12], les circuits qui gèrent l'ordre d'exécution sont représentés par le signet "->" sur le bord droit de la *partie "opérations"* de la MEMOIRE CENTRALE. Pratiquement, on fera avancer le signet de ligne en ligne dans le texte de l'algorithme. Le signet avance chaque fois à la fin de l'exécution de l'ordre décrit sur la ligne. Les ordres sont soit des opérations soit des instructions concernant la réservation préliminaire des cases de type "informations".

L'exécution de l'algorithme nécessite trois cases de la partie "informations" de la MEMOIRE CENTRALE de noms a, b et c dans lesquelles on pourra ranger des valeurs réelles (dans notre scénario, ces valeurs seront respectivement 2.6 pour a, 3.3 pour b et 5.9 pour c).

PREMIER PROGRAMME

Deux Exécutions

Exécution avec le *langage formel* (*Fig. 1.10a* à *Fig. 1.10e*) :

---

(12) Cf. la section 1.4.1.2. de ce Chapitre.

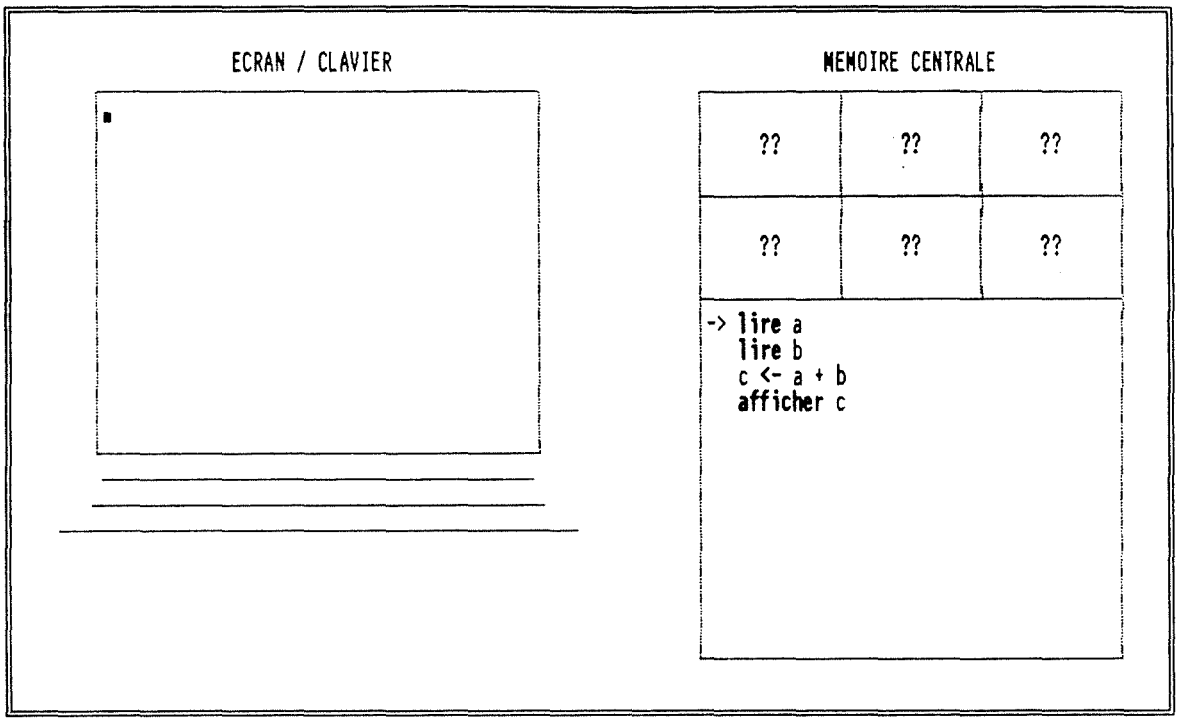


Fig. 1.10a Situation au démarrage de l'exécution du programme

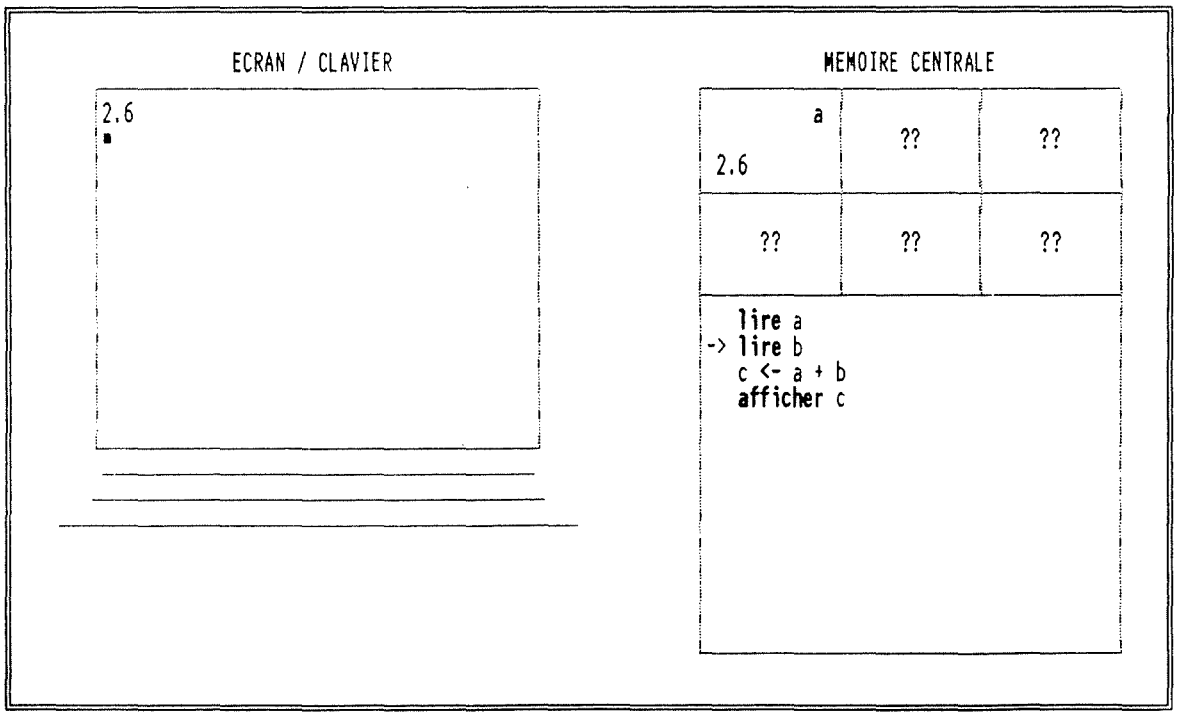


Fig. 1.10b Situation après l'exécution de la première opération de lecture.

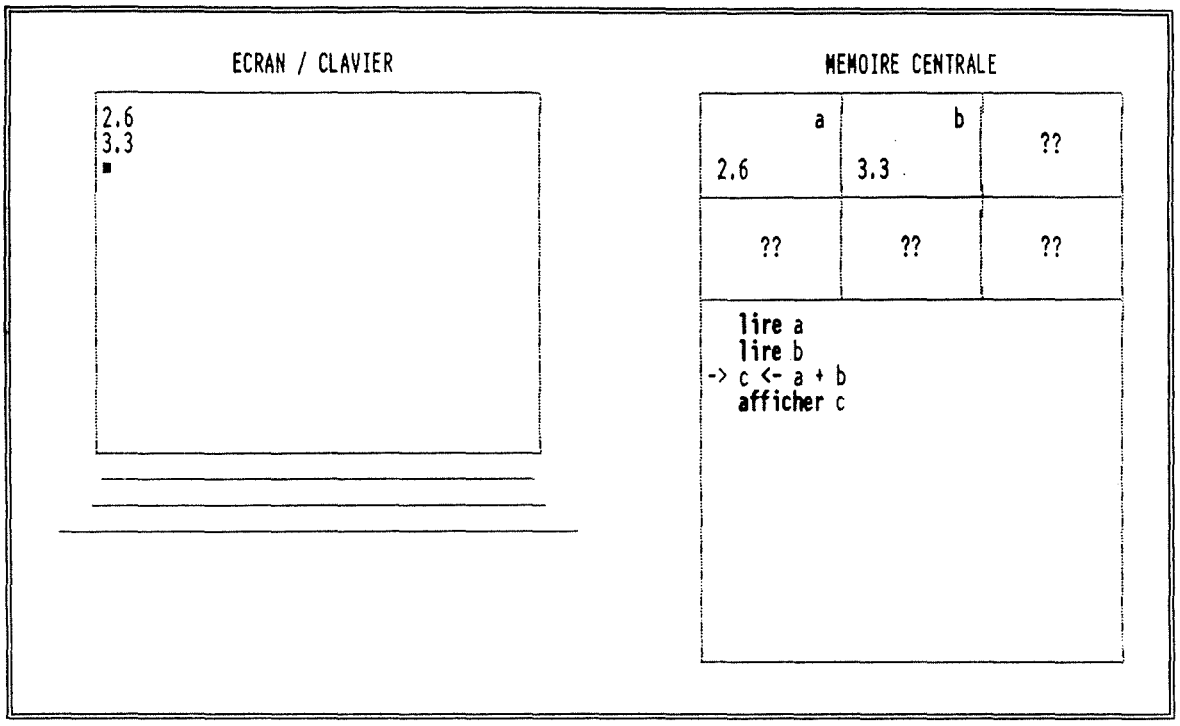


Fig. 1.10c Situation après l'exécution de la deuxième opération de lecture

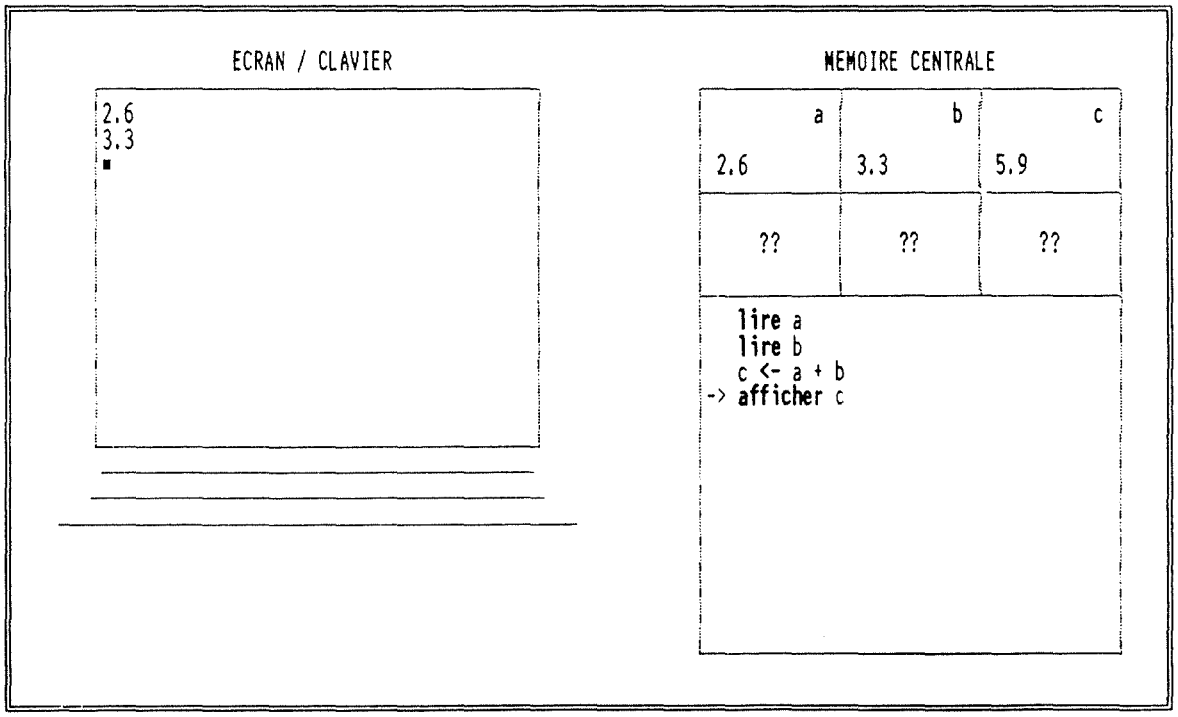


Fig. 1.10d Situation après l'exécution de l'opération d'affectation

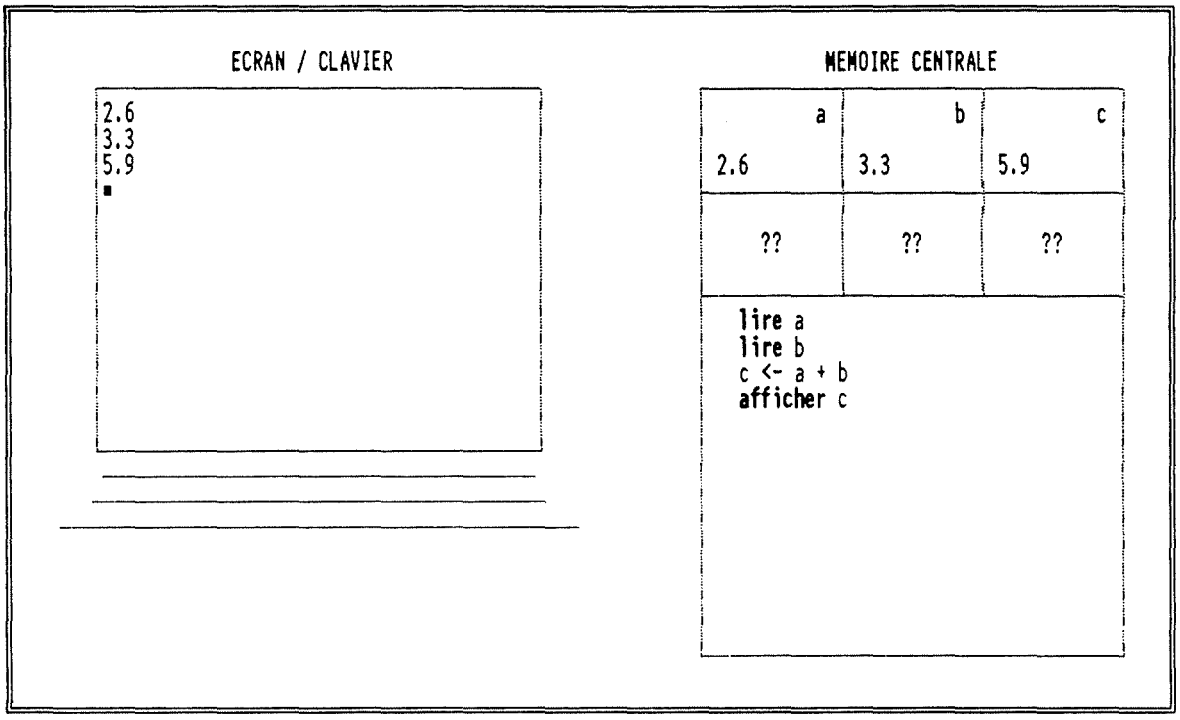


Fig. 1.10e Situation après l'exécution de l'opération d'écriture et situation en fin d'exécution du programme

Exécution avec le *langage Pascal* (*Fig. 1.11a* à *Fig. 1.11j*) :

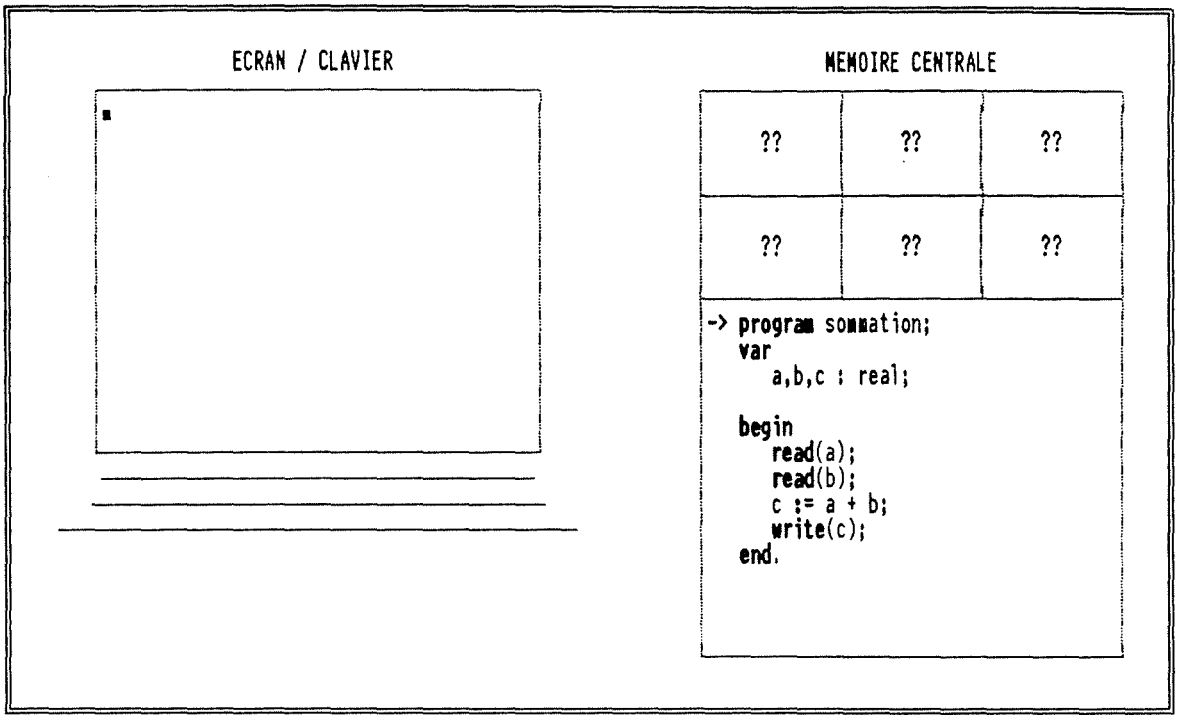


Fig. 1.11a Situation au démarrage de l'exécution du programme

On remarque qu'en *langage Pascal*, tout programme doit avoir un nom, ce qui se fait par l'ordre **program** suivi du nom du programme.

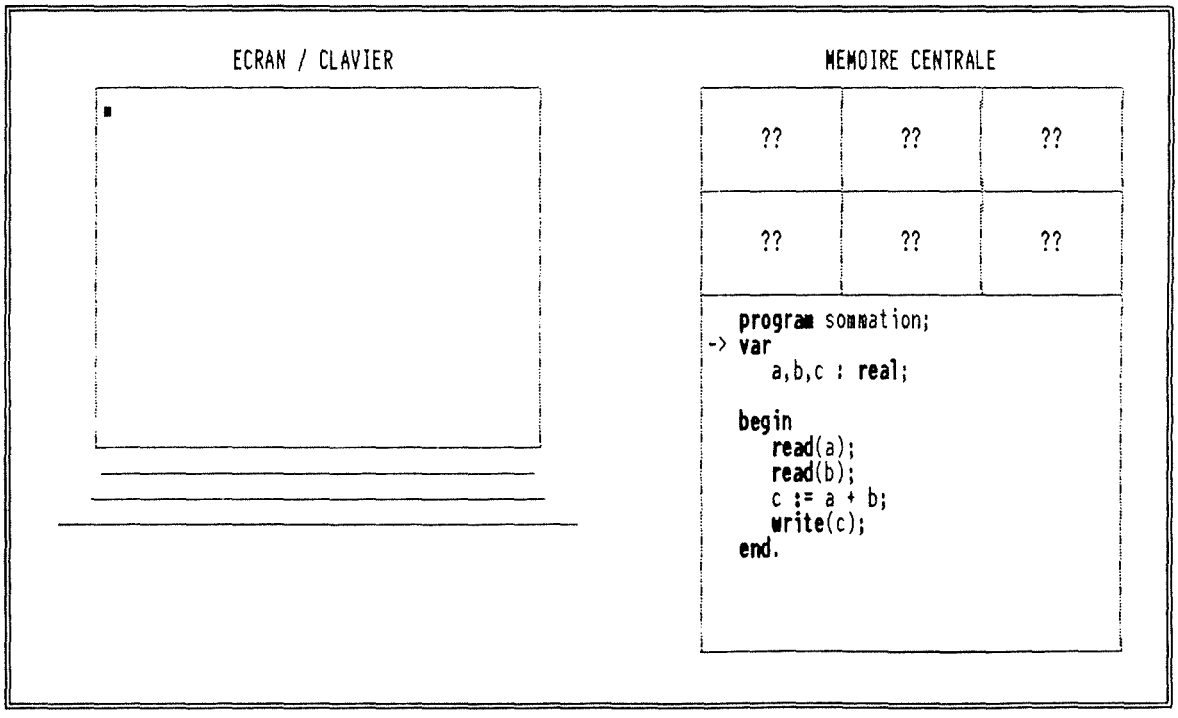


Fig. 1.11b Situation après la mémorisation du nom du programme

Le terme **var** est utilisé en *langage Pascal* pour indiquer la réservation de cases de type "informations".

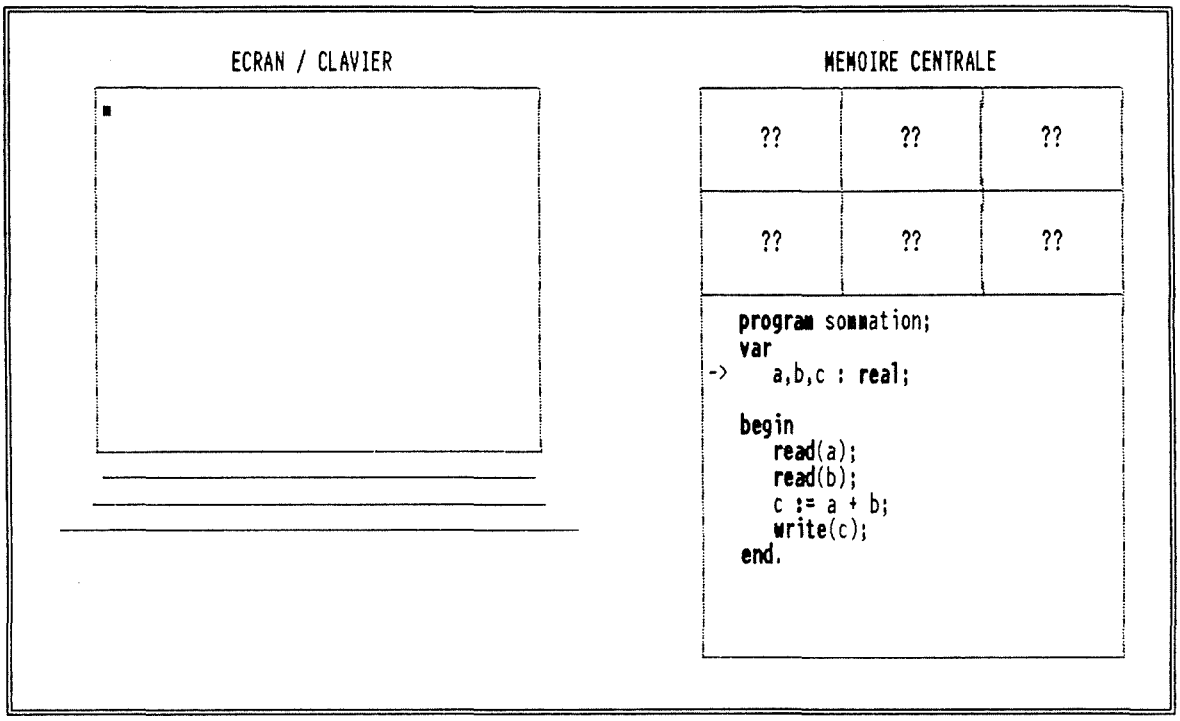


Fig. 1.11c Situation au début de la réservation des cases de type "informations"

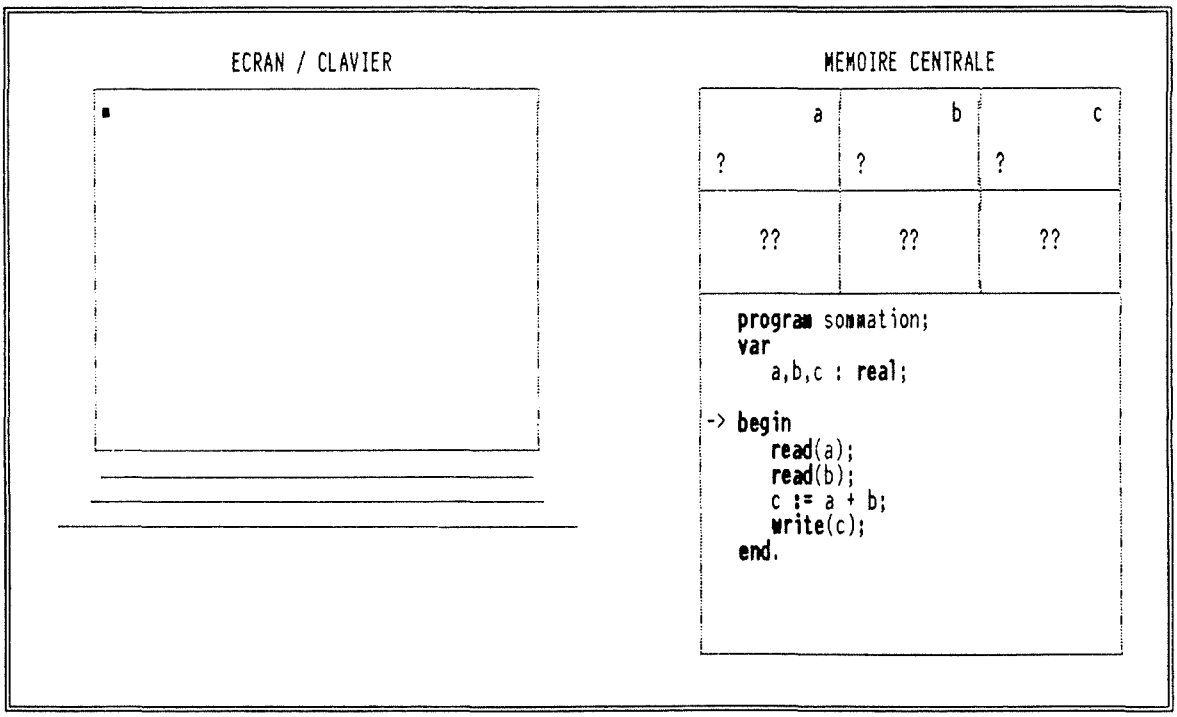


Fig. 1.11d Situation à la fin de la réservation de cases de mémoire

A ce moment, les trois premières cases "informations" ont un *nom* mais pas de *valeur* définie. Remarquons que l'exécution, à ce stade, d'une opération d'écriture produirait un résultat imprévisible! Avec le mot **begin** commence alors l'exécution proprement dite des opérations.

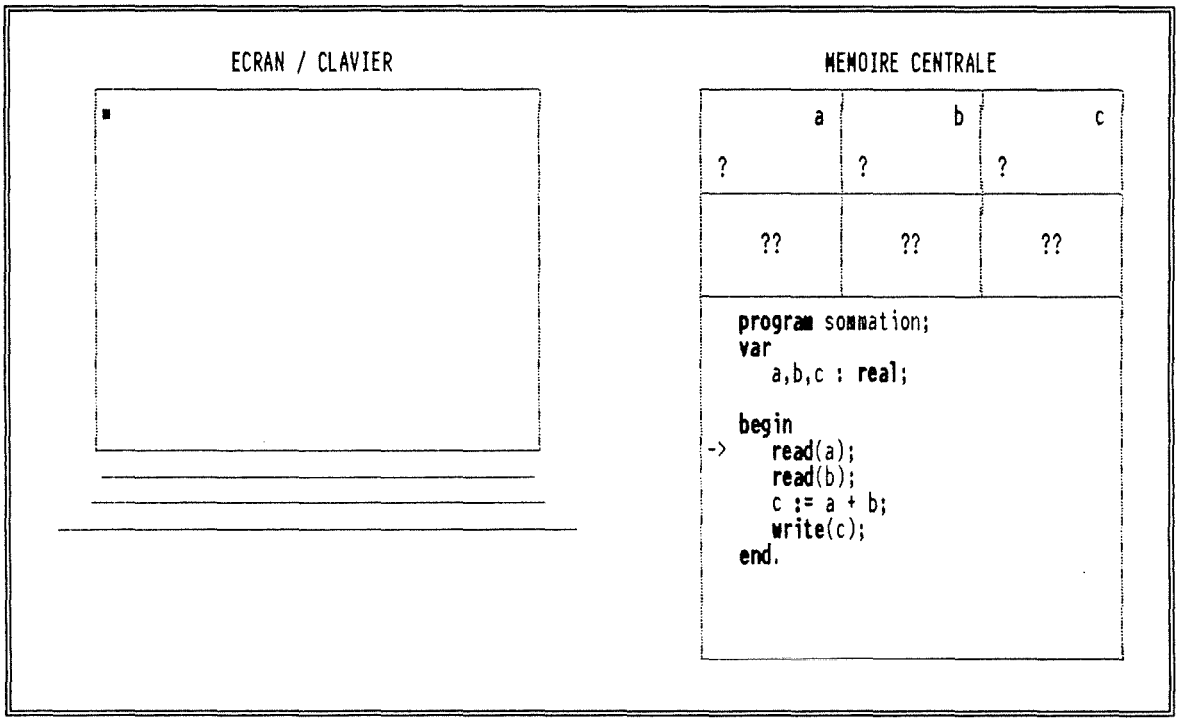


Fig. 1.11e Situation au début de l'exécution des opérations

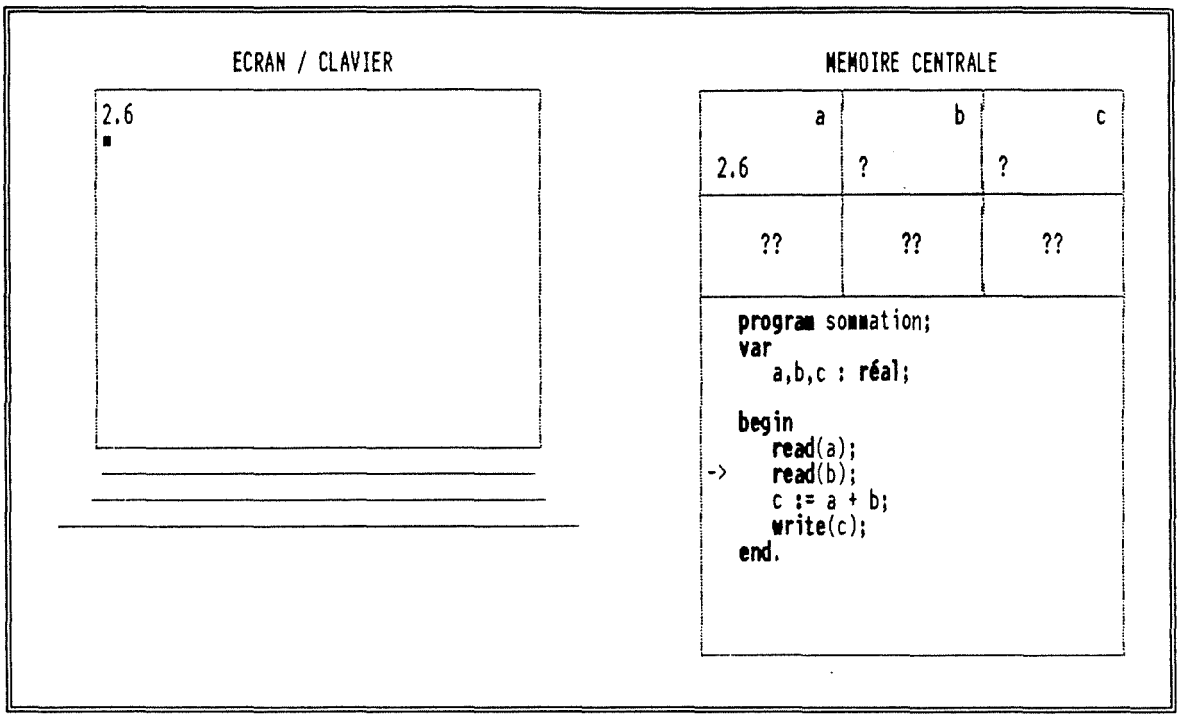


Fig. 1.11f Situation après la première opération de lecture

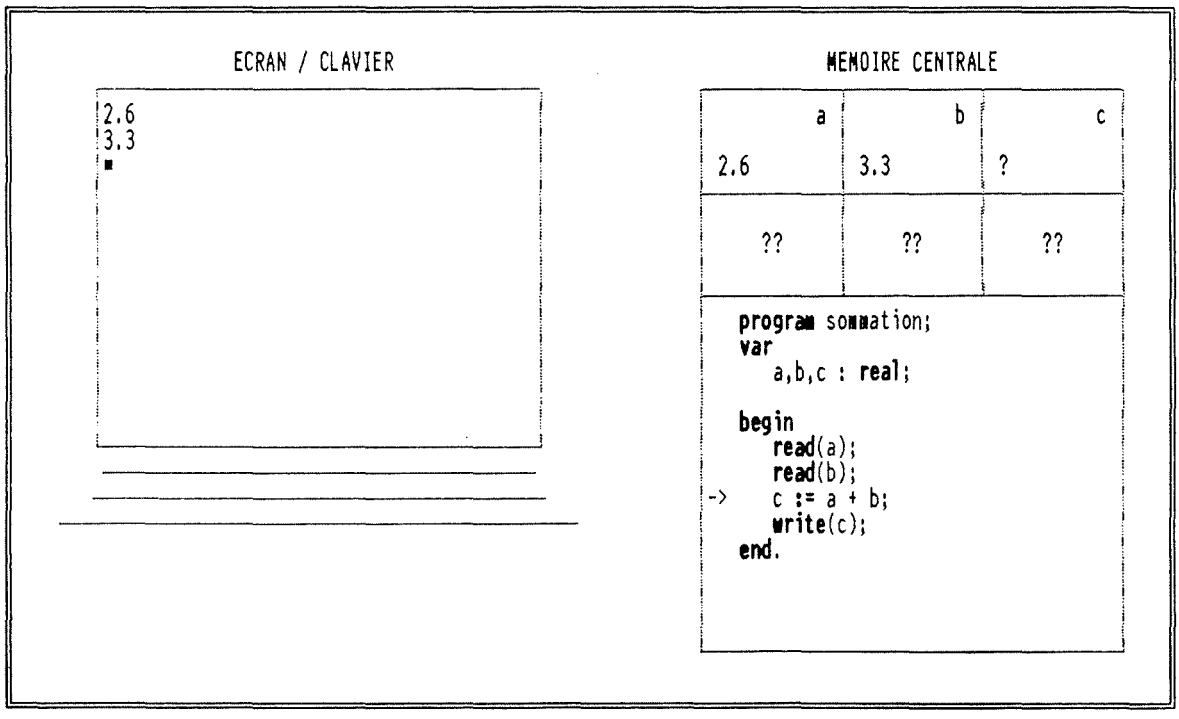


Fig. 1.11g Situation après la deuxième opération de lecture

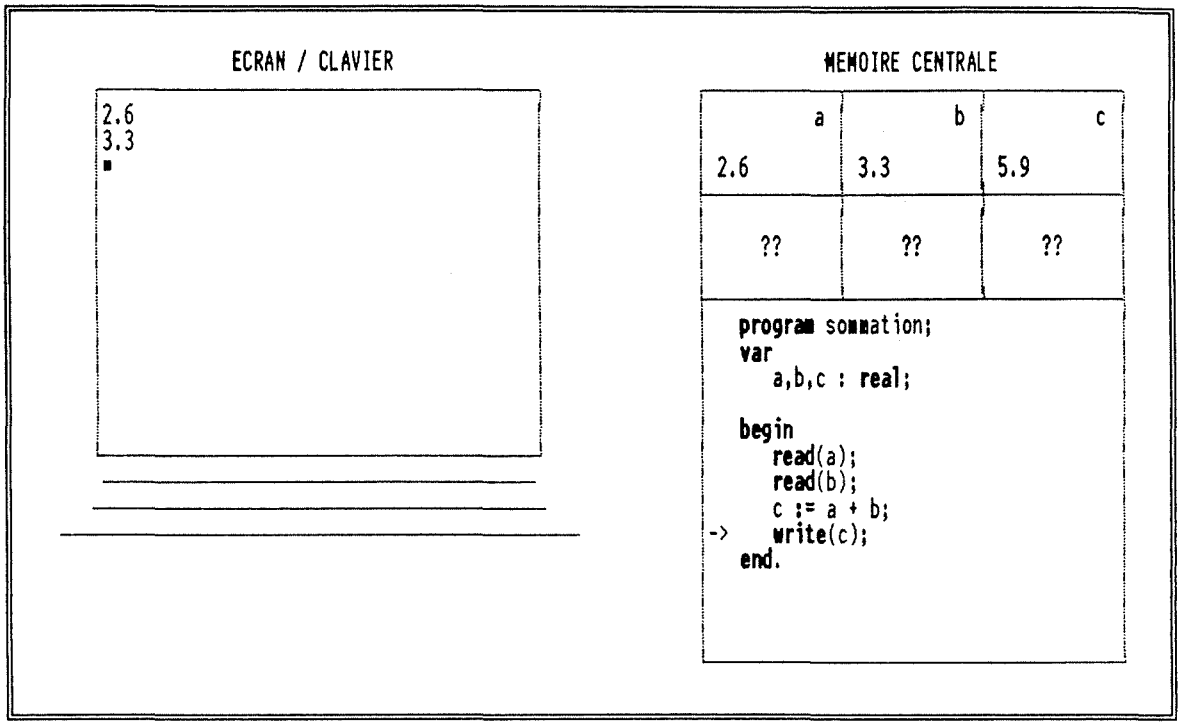


Fig. 1.11h Situation après l'exécution de l'opération d'affectation

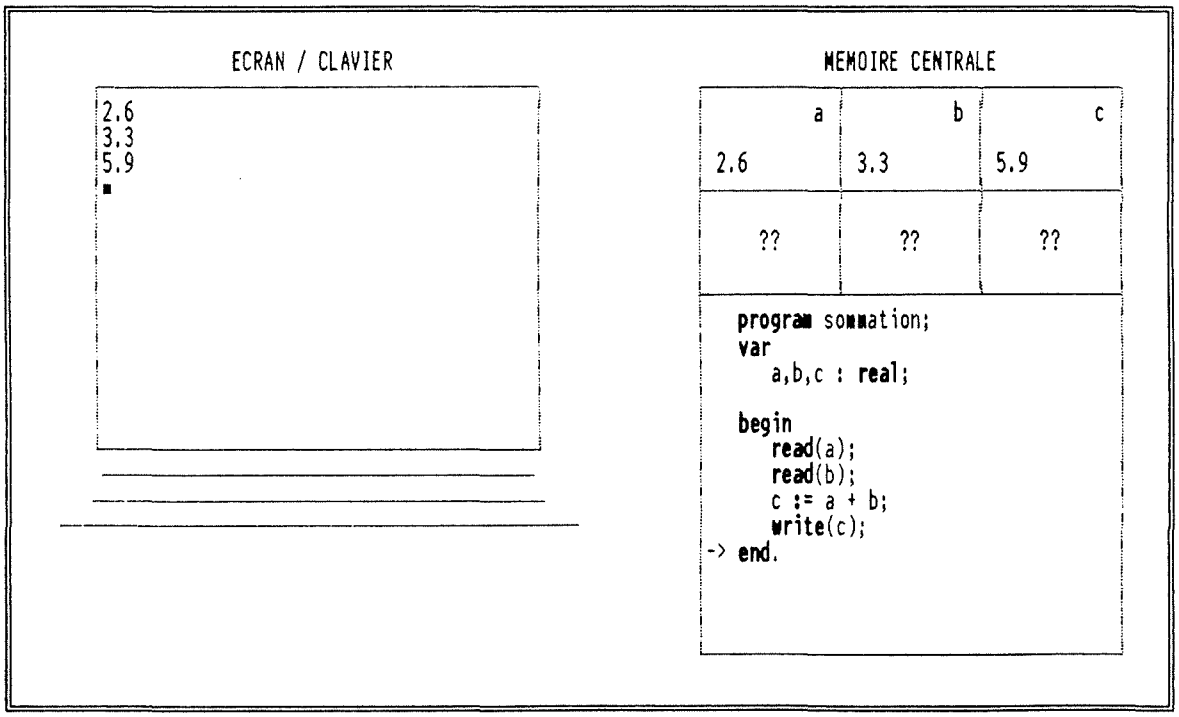


Fig. 1.11i Situation après l'exécution de l'opération d'écriture

Le mot **end** suivi du point indique à l'ordinateur que le programme s'achève.

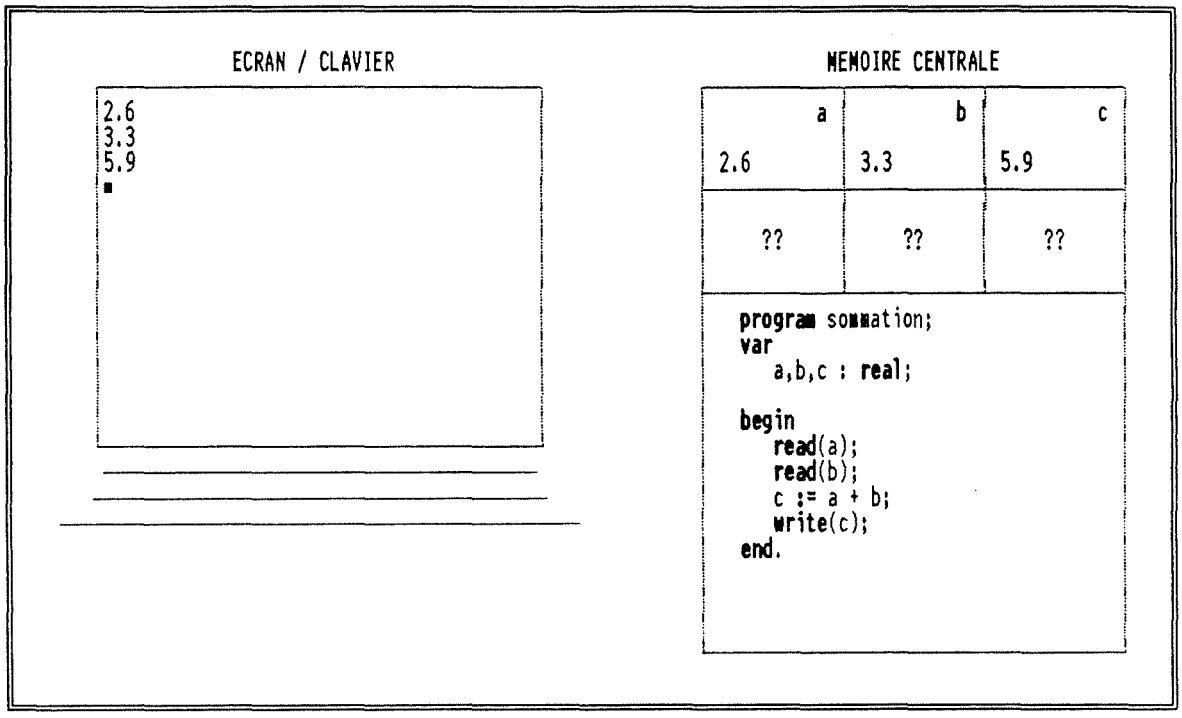


Fig. 1.11j Situation à la fin de l'exécution du programme

1.6 Conclusion.

Nous arrivons ainsi au terme de ce premier chapitre consacré à la présentation du **Modèle Général d'un Ordinateur** de Lesuisse et Borsu. Ce chapitre peut sembler long mais il nous est apparu essentiel d'isoler de la méthode d'*Initiation aux Raisonnements de la Programmation* de Lesuisse et Borsu [A/LE87] tout ce qui concernait leur **Modèle Général d'un Ordinateur** et de le mettre en exergue à ce travail. Comme on va s'en rendre compte, ce modèle constitue en effet le pivot du mémoire.

1.7 Sources bibliographiques.

Ce premier chapitre est essentiellement basé sur l'ouvrage de Lesuisse et Borsu donné dans la section

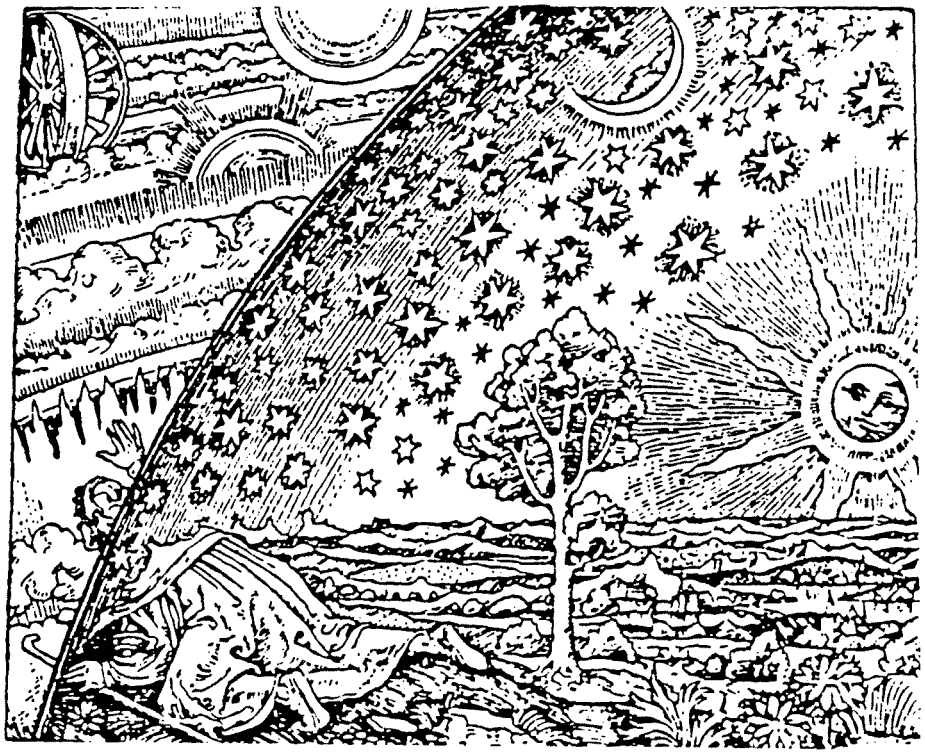
A. Ouvrage de Référence

de la Bibliographie.

Cependant, le travail de ces deux auteurs s'intègre dans un courant d'idée visant à développer des méthodologies pour un apprentissage plus performant (qui dépasse d'ailleurs le seul domaine de la Programmation). Aussi, pour trouver plus d'informations sur les idées sous-jacentes au contenu de ce chapitre, nous avons consultés un certain nombre d'articles rassemblés dans la section

B. Apprentissage de la Programmation

de la bibliographie. Notons que plusieurs de ces travaux se trouvent regroupés dans un recueil édité en 1986 par B. Curtis pour la Computer Society sous le titre *"Tutorial: Human Factors in Software Development"*.



Le système du monde (*Frontispice du Moyen Age*)

Chapitre 2

CONCEPTION D'UN OUTIL INFORMATIQUE ANNEXE

2.1 Introduction.

Dans ce second chapitre, nous examinons sur un exemple les différentes étapes de la conception d'un outil informatique.

Ce chapitre fait suite au premier chapitre à deux titres:

- D'une part, la conception d'un outil informatique n'est autre qu'une spécialisation de ce qui a été décrit dans le premier chapitre à propos de la résolution d'un problème, ce qui permet d'observer les raffinements à apporter au schéma de base en trois étapes lorsque le problème à résoudre devient un projet.
- D'autre part, l'outil considéré consiste à informatiser le **Modèle Général d'un Ordinateur** de Lesuisse et Borsu présenter au premier chapitre ainsi que son utilisation pour illustrer comment un ordinateur procède pour résoudre un problème,

L'objectif poursuivi dans ce chapitre consiste à faire ressortir la démarche poursuivie pour la réalisation de l'outil en continuité avec le premier chapitre. Il ne nous a malheureusement pas été permis de décrire toutes les étapes dans les détails.

Nous commençons par présenter le projet de l'outil informatique tel qu'il nous a été présenté. Après un commentaire général à propos du passage de la résolution d'un problème à la conception d'un projet, nous donnons alors une

définition pour le cadre général de l'outil. Ensuite, nous spécifions les fonctionnalités de l'outil par la description des principes de fonctionnement, par la définition des objets à manipuler et des résultats à obtenir et enfin, par l'énumération des fonctions nécessaires au traitement. Après, nous exposons et justifions l'Architecture Logique donnant une description abstraite de la répartition du travail en unités modulaires autonomes.

2.2 Présentation du projet d'un outil informatique.

2.2.1 Le point de départ.

Dans le premier chapitre, nous avons présenté le **Modèle Général de l'Ordinateur** de Lesuisse et Borsu [A/LE87] ainsi que les principes du fonctionnement d'une résolution de problème dans le cadre du modèle illustrés sur un problème élémentaire. Cette résolution a été exposée avec la "*feuille de papier*" comme *support de représentation*. Il est clair qu'une telle procédure n'est pas d'un maniement aisé. Actuellement, Lesuisse et Borsu utilisent leur **Modèle** à des fins d'enseignement afin d'introduire de nouveaux concepts. Pour les expérimentations et les exercices faits par les étudiants eux-mêmes, ils proposent un schéma simplifié en colonnes(1) plus facile d'emploi sur papier.

On imagine aisément que le **Modèle Général d'un Ordinateur** pourrait être exploité beaucoup plus efficacement si les étudiants avaient à leur disposition un outil informatique utilisant l'*ordinateur*, lui-même, comme *support de représentation*. C'est ainsi qu'il nous a été demandé d'examiner comment réaliser un tel outil informatique(2).

---

(1) On trouvera les détails sur ce schéma à la page 87 de l'ouvrage de Lesuisse et Borsu [A/LE87].

(2) Soulignons que la demande de réalisation de cet outil informatique remonte au mois de septembre 1987 et que la majeure partie du développement qui en a résulté a été effectuée durant l'année académique 1987-1988.

2.2.2 Intérêt et originalité de l'outil.

Avec un tel outil, le **Modèle Général d'un Ordinateur** prendrait toute sa dimension pédagogique et cela permettrait de situer la *Méthode d'Initiation aux Raisonnements de la Programmation* de Lesuisse et Borsu [A/LE87] dans une conjoncture d'Enseignement Assisté par Ordinateur (EAO). En effet, l'étudiant serait alors capable de faire ses propres expérimentations "à la carte" dans le cadre du modèle et avec l'ordinateur comme assistant.

L'originalité de l'outil projeté réside dans le fait que, dans ce cas, l'ordinateur permet d'assister un enseignement sur l'ordinateur. Il y a *interconnexion entre l'outil qui sert à apprendre et l'outil qu'il faut apprendre*.

2.2.3 Conditions pour un bon outil.

Suite à la première discussion que nous avons eue avec les concepteurs du Modèle, nous avons pu dégager un certain nombre de conditions que l'outil devrait vérifier pour qu'il puisse répondre à leur attente. Nous avons retenu comme capitales les six conditions suivantes:

1) *Reproduire* le plus *fidèlement* possible la schématisation du **Modèle Général de l'Ordinateur** présenté au premier chapitre(3).

2) Permettre l'*exécution d'algorithmes* construits par les étudiants eux-mêmes.

3) Prendre en compte des algorithmes dont la *longueur du texte* ainsi que le *nombre des informations et résultats* intervenant dans le problème *ne soient pas limités* par les dimensions "physiques" de l'écran de l'ordinateur.

---

(3) Ceci consiste à reproduire au mieux le schéma de la figure *Fig. 1.8* du chapitre 1.

4) Prévoir la possibilité d'exécuter des *algorithmes* écrits dans *divers langages de programmation*, y compris dans un *langage formel* de construction d'algorithmes.

5) *Autoriser* les différents *modes de raisonnement* décrits dans la Méthode d'Initiation aux Raisonnements de la Programmation de Lesuisse et Borsu [A/LE87] avec *gestion* appropriée du *signet* indiquant l'opération en cours d'exécution.

6) *Utiliser le matériel ordinateur* (hardware) mis à la disposition des étudiants et concevoir l'outil dans un *environnement* le plus proche possible des *outils logiciels* mis à leur disposition(4)

2.2.4 Souhaits pour la réalisation.

Les concepteurs du Modèle ont également émis des souhaits quant à la façon dont ils voulaient que le travail soit entrepris. Cela concernait les trois points suivants:

- D'une *manière générale*:
  - adopter une *découpe* du travail la plus *modulaire* possible pour bien sérier les différents problèmes à résoudre et autoriser des améliorations et extensions ultérieures.

- Pour la *reproduction du modèle sur l'écran*:
  - utiliser un système de *multi-fenêtrage* avec une fenêtre plein écran pour représenter l'ECRAN du **Modèle Général d'un Ordinateur** et une fenêtre en sur-impression pour représenter la MEMOIRE CENTRALE.

- Pour l'*organisation* et l'*édition* de la MEMOIRE CENTRALE:
  - dissocier les dimensions de la MEMOIRE CENTRALE et de l'écran "physique" en faisant appel aux principes des systèmes d'éditions interactives comme on les utilisent

---

(4) en l'occurrence des PC Olivetti M24 (type 8086) avec le compilateur *Turbo Pascal* de Borland 2.0 ou 3.0.

dans les logiciels de traitement de texte et tels qu'ils sont décrits dans l'article de Meyrowitz et Van Dam [C/ME82].

2.2.5 Choix d'un nom.

Avant d'exposer les modalités du développement de l'outil informatique projeté, nous convenons de lui donner un *nom*. Comme indiqué auparavant, celui-ci se présente comme un outil annexe à la *Méthode d'Initiation aux Raisonnements de la Programmation* de Lesuisse et Borsu [A/LE87]. C'est pourquoi, nous proposons de retenir le vocable **PROGRAIS**, abréviation de "**PROG**rammation par le **RAIS**onnement", puisqu'il est destiné à contribuer aux *progrès* réalisés par les étudiants dans leur apprentissage de l'informatique.

2.3 Du problème au projet.

La démarche à suivre pour le *développement d'un projet informatique* est foncièrement comparable à celle qui a été décrite au premier chapitre pour la résolution d'un problème. On retrouve toujours les trois grandes étapes de toute activité de programmation, qui sont :

1° définir le cadre général,
2° construire la marche à suivre,
3° procéder à l'exécution.

La manière de satisfaire à ces tâches est naturellement plus complexe. Cela provient tout d'abord du fait qu'on n'est plus en présence d'un problème formant un tout mais plutôt, d'un *système constitué de plusieurs sous-problèmes* différents reliés entre eux. Il faut donc pouvoir spécifier toutes les fonctionnalités et isoler les différents sous-problèmes avant de construire la marche à suivre. De plus, dans le cas d'un projet, le travail de développement n'est plus toujours l'oeuvre d'une seule personne. Il faut donc faire en sorte que puisse s'instaurer un travail d'équipe efficace en établissant des conventions communes de représentation et une répartition des différentes tâches. Une autre difficulté peut également

provenir du fait que l'élaboration du projet n'est plus toujours situé dans un lieu géographique unique. Il faut donc également veiller à s'extraire le plus possible du matériel par une approche abstraite.

L'analyse de tous les raffinements à apporter au schéma de base en trois étapes décrit dans le premier chapitre constitue un part importante de l'activité de programmation avancée. En effet, comme on peut s'en douter, les qualités d'un projet informatique dépendent fortement de la bonne conduite de son développement. Tout cela relève de la méthodologie de développement de logiciels, domaine appelé en anglais "Software Engineering", qui étudie les techniques permettant d'améliorer la fiabilité, la maintenance et la performance des gros projets informatiques [D/VA87] [D/ME88].

Le projet proposé dans le cadre de ce mémoire reste un projet informatique de petite dimension. Cependant, il constitue un bon exemple de mise en oeuvre de certains raffinements résultant d'une démarche de développement basée sur les techniques actuelles de développement de logiciels. En appliquant ces techniques, on aura aussi satisfait au premier souhait de Lesuisse et Borsu(5), à savoir une découpe du travail la plus modulaire possible et susceptible de modifications et améliorations.

Avec les raffinements retenus, le schéma de base en trois étapes est modifié dans un nouveau schéma dont les étapes sont respectivement:

1° définir le *cadre général*, ce qui revient à faire l'analyse des besoins,

2° faire une *analyse fonctionnelle* qui reprend les principes de fonctionnement, les objets à manipuler, les résultats à produire ainsi que les différents fonctions nécessaires au traitement.

---

(5) Cf. section 2.2.4 de ce chapitre.

3° construire une *architecture logique* en établissant une découpe abstraite du travail suivant les problèmes spécifiques,

4° construire une *architecture physique* qui donne une vue concrète de la marche à suivre,

5° établir un plan de développement par incréments successifs permettant d'isoler des sous-sytèmes autonomes intermédiaires,

et enfin, pour chaque sous-système retenu,

6° faire le codage,

7° procéder à l'exécution,

avec en fin de travail,

8° produire un outil informatique utilisable.

Dans la suite du chapitre, nous allons passer en revue les trois premières étapes sans toutefois entrer toujours dans les détails. Pour les étapes suivantes nous nous limitons de les illustrer au moyen du codage proposé dans l'annexe 2.

2.4 Le cadre général de l'outil.

La première étape du développement consiste encore à *énoncer* ou *spécifier* le cadre général du système informatique. Cela revient à faire l'analyse des besoins. En fait, c'est ce qui a déjà été fait dans la première section de ce chapitre. On peut donc considérer que cette section constitue le cadre général de l'outil. Cependant, s'il fallait résumer le cadre général en une seule phrase, on pourrait écrire:

"On demande de réaliser un outil informatique annexe à la Méthode d'Initiation aux Raisonnements de la Programmation de Lesuisse et Borsu [A/LE87] capable d'illustrer dans le cadre de leur Modèle Général de l'Ordinateur comment l'ordinateur procède pour exécuter un algorithme associé à un problème."

Remarquons que l'outil à réaliser n'est rien d'autre qu'un *Compilateur-Interpréteur* de programme évolué. La particularité supplémentaire par rapport à un Compilateur-Interpréteur classique réside dans l'incorporation d'une représentation à l'écran de la face cachée de l'ordinateur en suivant les principes du **Modèle Général d'un Ordinateur** de Lesuisse et Borsu.

2.5 L'analyse fonctionnelle.

2.5.1 Présentation.

Le cadre général ne peut donner qu'une idée très vague sur la façon de concevoir l'outil. Cette première description est affinée à l'aide de l'analyse fonctionnelle. Dans cette analyse, on spécifie plus en détails:
- les principes retenus pour le fonctionnement,
- les objets à manipuler,
- les résultats à produire

ainsi que
- les fonctions nécessaires au traitement.

2.5.2 Les principes de fonctionnement.

2.5.2.1 *Représentation sur l'écran "physique".*

Afin de répondre au mieux au deuxième souhait des concepteurs(6), nous proposons de représenter les éléments du **Modèle Général d'un Ordinateur** selon les principes schématisés dans la figure *Fig. 2.1*:

---

(6) Il s'agit des souhaits exprimés dans la section 2.2.4 de ce chapitre.

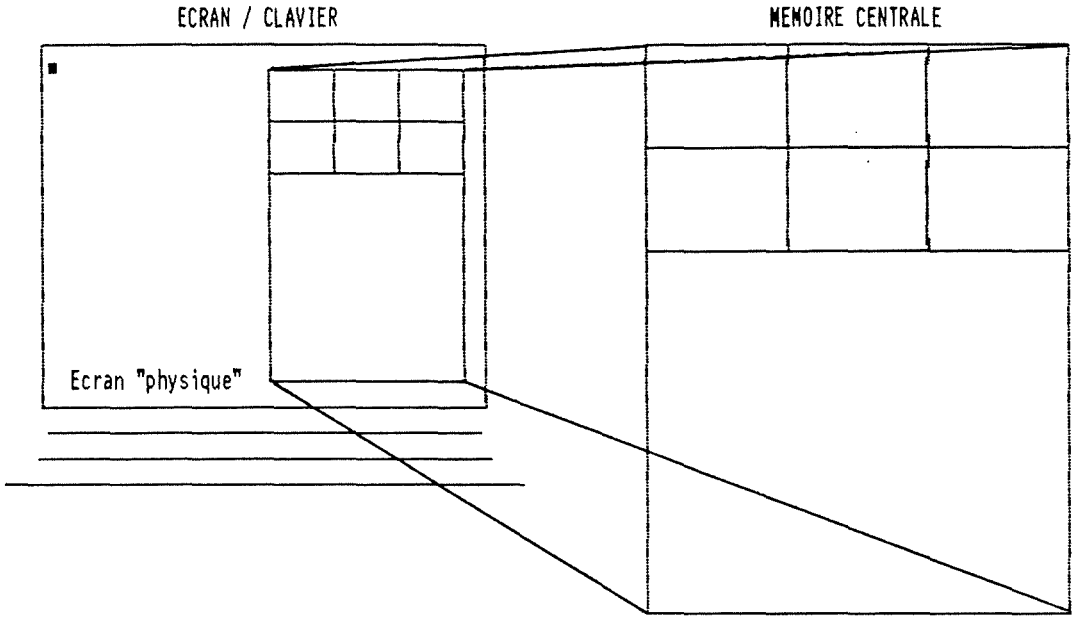


Fig. 2.1 Principes de la représentation en multi-fenêtrage du Modèle Général de l'Ordinateur

L'ECRAN/CLAVIER du modèle se confond avec l'écran "physique" et le clavier "physique" de l'ordinateur plaçant l'utilisateur dans les mêmes conditions que s'il utilisait un outil logiciel classique. La MEMOIRE CENTRALE s'affiche en sur-impression sur la moitié droite de l'écran "physique" afin de ne pas occulter la partie de l'écran la plus à même d'être utilisée.

2.5.2.2 *Organisation et édition de la Mémoire Centrale.*

Quant à la façon d'organiser et d'éditer la MEMOIRE CENTRALE, nous pensons satisfaire les concepteurs (troisième souhait) en travaillant de la manière suggérée dans la figure suivante (*Fig. 2.2*):

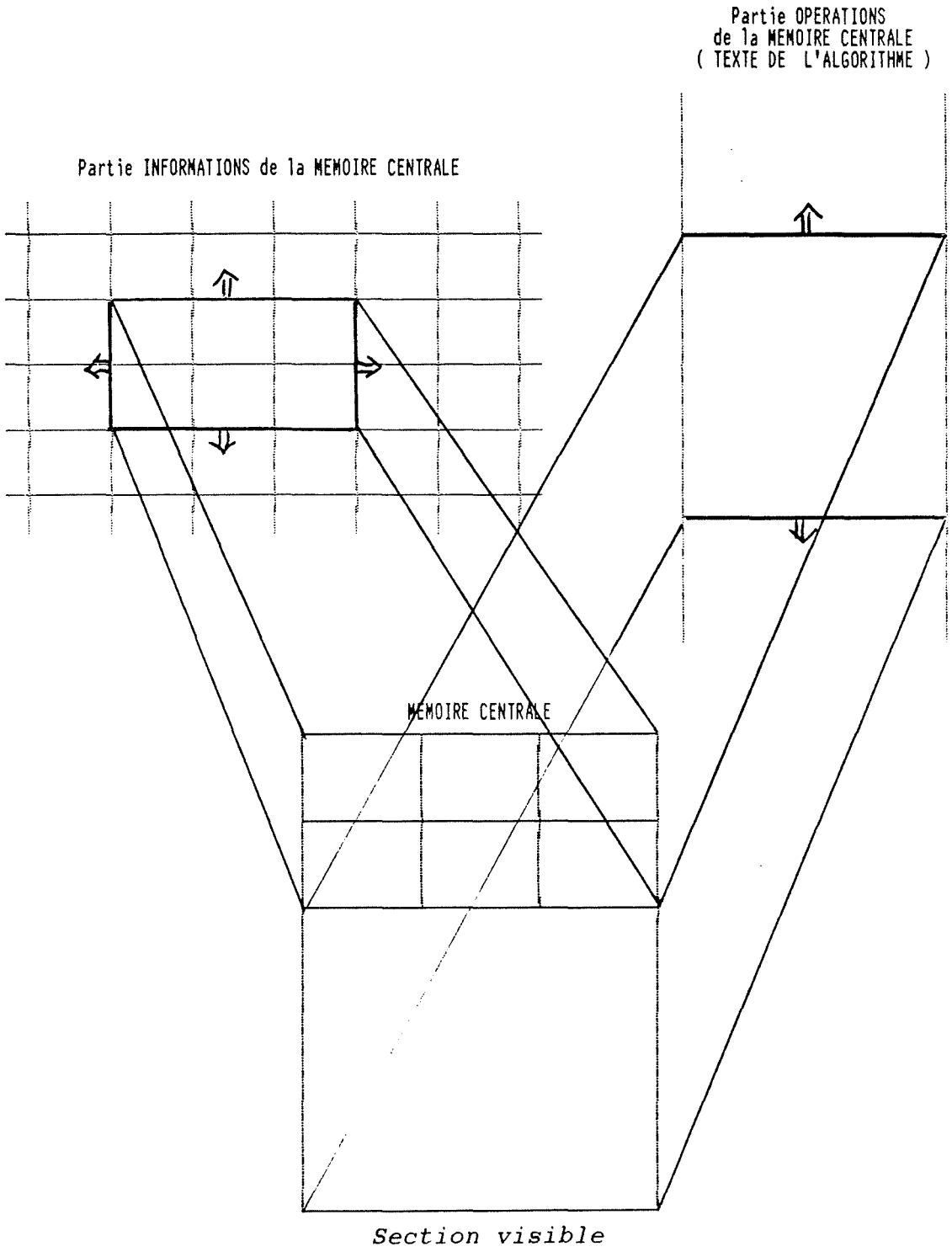


Fig. 2.2 Principes de l'édition de la Mémoire Centrale
(les flèches indiquent les déplacements des sections visibles)

La section de la MEMOIRE CENTRALE visible à l'écran se présente comme une lucarne permettant de visionner en même temps six cases contiguës de la partie INFORMATIONS ainsi

qu'un nombre prédéfini de lignes du texte de l'algorithme contenu dans la partie OPERATIONS.

Le déplacement des deux lucarnes se fait en fonction de l'exécution du programme afin de visualiser la section du texte contenant la ligne en cours de traitement ainsi que la section de la partie INFORMATIONS où s'effectue des modifications telles qu'une nouvelle réservation de case ou l'introduction d'une nouvelle valeur dans une case déjà réservée. Pour pouvoir mieux situer la ligne en cours de traitement dans son contexte, le déplacement dans la partie OPERATIONS s'effectue de façon à ne pas placer cette ligne en haut de la lucarne.

On peut observer sur la figure que les déplacements dans la partie OPERATIONS se font uniquement verticalement. A posteriori, nous constatons que ce choix est limitatif parce qu'il lie la largeur effective du texte du programme à la largeur "physique" de la Section visible de la MEMOIRE CENTRALE. Cela illustre l'importance de faire les bons choix déjà au niveau de l'analyse fonctionnelle.

2.5.2.3 *Edition dans une case de la partie INFORMATIONS.*

Pour l'édition d'une case de la partie INFORMATIONS de la MEMOIRE CENTRALE, nous avons adoptés les principes suivants (*Fig. 2.3*):

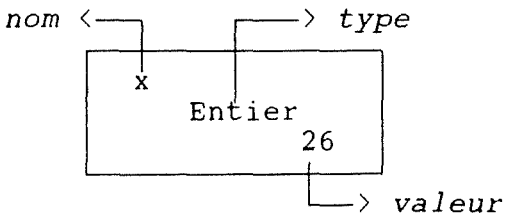


Fig. 2.3 Principes de l'édition d'une case de la partie INFORMATION
(case de nom "x", de type entier et de valeur "26")

Ces conventions diffèrent légèrement des conventions adoptées par Lesuisse et Borsu [A/LE87][7]. Le *nom* est placé en haut et à gauche pour pouvoir disposer de toute la largeur de la case. Il faut, en effet, tenir compte du fait que la largeur de la case limite le nombre de caractères visible sur l'écran "physique". La *valeur* est placée en bas et à droite de la case. Nous avons convenus d'ajouter également au milieu le *type* associé à la case.

2.5.2.4 *Edition de l'opération en cours d'exécution.*

Pour l'édition du *signet* qui permet de représenter les circuits qui gèrent l'ordre d'exécution, nous convenons d'utiliser une des possibilités que donne l'ordinateur et qui consiste à placer l'opération en cours d'exécution en image "Vidéo Inversée".

2.5.2.5 *Conception de l'exécution du programme.*

Il existe deux façons de concevoir l'exécution d'un programme, soit par compilation, soit par interprétation.

- Par compilation, le programme, appelé programme source, est traduit dans un programme équivalent, appelé programme objet, écrit dans le langage connu de l'ordinateur ou l'exécution à lieu et c'est ce programme objet qui est alors exécuté par l'ordinateur.
- Par interprétation, le programme, appelé programme source, n'est pas traduit et il est directement exécuté via un interpréteur. L'ordinateur examine alors chaque opération les unes après les autres et les exécute directement.

Vu le caractère interactif exigé par la réalisation de l'exécution de programme dans le cadre du **Modèle Général d'un Ordinateur**, nous pensons que c'est le mode par interprétation qui s'impose pour l'outil. Ce mode permet en effet une flexibilité beaucoup plus grande du système.

---

(7) Ces conventions sont reprises dans la figure *Fig. 1.4* du premier chapitre.

2.5.3 Les objets à manipuler.

La mise en oeuvre de l'outil suivant les principes qui viennent d'être énoncés nécessite la manipulation d'un certain nombre d'objets exactement comme cela était le cas pour la résolution du premier problème du premier chapitre. Au stade d'une analyse fonctionnelle, les objets sont des *objets abstraits* qui permettent de spécifier les règles de fonctionnement de l'outil.

Nous choisissons tout d'abord quatre objets relatifs aux concepts de base du **Modèle Général d'un Ordinateur** de Lesuisse et Borsu, à savoir:

- OBJET : ***Algo***

  Définition:

  Programme (ou algorithme) qui consiste en une construction permettant de résoudre un problème et qui est composée d'une suite chronologique d'opérations primitives à exécuter, le tout étant exprimé dans un langage de programmation adéquat.

  Remarque:

  Il s'agit ici du programme (ou algorithme) abstrait, à ne pas confondre avec la représentation concrète de ce programme dans un langage de programmation(8).

- OBJET : **MC:Op**

  Définition:

  Partie le la Mémoire Centrale de l'ordinateur capable de mémoriser un programme (ou algorithme).

  Remarque:

  Cet objet représente la partie OPERATIONS de la MEMOIRE CENTRALE du **Modèle Général d'un Ordinateur** capable de mémoriser le programme dans son entièreté.

---

(8) Cette différence de point de vue sera explicitée plus en détails dans le chapitre 3.

- OBJET : **MC:Inf**

  Définition:

  Partie de la Mémoire Centrale de l'ordinateur capable de mémoriser les informations nécessaires pour l'exécution d'un programme (ou algorithme) ainsi que les résultats qui découlent de son exécution.

  Remarque:

  Cet objet représente la partie INFORMATIONS de la MEMOIRE CENTRALE du **Modèle Général d'un Ordinateur** capable de mémoriser toutes les informations et résultats nécessaires au programme.

- OBJET : **EC**

  Définition:

  Système formé d'un écran et d'un clavier permettant d'assurer les échanges avec l'utilisateur lors de l'exécution d'un programme (ou algorithme).

  Remarque:

  Cet objet représente le système ECRAN/CLAVIER du **Modèle Général d'un Ordinateur.**

Il est bien entendu que les trois derniers objets ne font pas référence à la représentation sur l'écran "physique" de la machine "ordinateur" utilisée par l'outil. Pour cela, nous choisissons deux objets supplémentaires. Il s'agit de:

- OBJET : **ImageMC**

  Définition:

  Section de la MEMOIRE CENTRALE visible sur l'écran "physique".

- OBJET : **ImageEC**

  Définition:

  Image de l'écran donnant le retour sur l'écran "physique" de l'ECRAN du **Modèle Général d'un Ordinateur.**

Pour la gestion proprement dite de l'exécution du programme, nous choisissons un dernier objet, à savoir:

- OBJET : ***symbole***

  Définition:

  Symboles appartenant ou n'appartenant pas au langage de programmation reconnu par l'outil.

  Remarque:

  Il s'agit ici de la notion abstraite d'un symbole(9).

2.5.3 Les résultats.

Les objets définis dans la section précédente forment l'ensemble des objets abstraits nécessaires pour le fonctionnement de l'outil. Les fonctions opèrent alors sur ces objets pour fournir les résultats.

2.5.4 Les fonctions.

La construction de l'outil se résume en une suite chronologique de manipulation sur les objets définis. Nous allons à présent préciser les manipulations que l'outil doit être capable d'effectuer sous forme de fonctions. Chaque objet possède les fonctions nécessaires à sa propre manipulation. Cette façon de faire est inspirée des méthodes de développement *Orienté Objet* ou *Types Abstraits*. En les rattachant chaque fois à l'objet concerné, ces fonctions sont respectivement

Sur l'OBJET ***Algo*** :

- FONCTION : Choisir_Algo

  Définition:

  Choisir le programme à exécuter.

- FONCTION : Test_Bon_Algo

  Définition:

  Vérifier si le programme est un algorithme exécutable (par rapport au choix du langage et au choix de mise en page).

- FONCTION : Executer_Algo

  Définition:

---

(9) Cette différence de point de vue sera explicitée plus en détails dans le chapitre 3.

Exécuter le programme.

- FONCTION : Executer_Op_Algo
  Définition:
  Exécuter une opération du programme.
- FONCTION : Test_Nouvelle_Op_Algo
  Définition:
  Tester la fin de l'exécution d'une opération.

Sur l'OBJET **MC:Op** :

- FONCTION : Creer_MC:Op_vide
  Définition:
  Créer **MC:OP**.
- FONCTION : Mettre_Algo_dans_MC:Op
  Définition:
  Modifier **MC:OP** conformément à la mémorisation du programme.
- FONCTION : Mettre_Signet_sur_Op
  Définition:
  Modifier **MC:OP** conformément à l'introduction du signet indiquant l'opération à exécuter.
- FONCTION : Enlever_Signet_sur_Op
  Définition:
  Modifier **MC:OP** conformément à la suppression du signet indiquant l'opération à exécuter.

Sur l'OBJET **MC:Inf** :

- FONCTION : Creer_MC:Inf_vide
  Définition:
  Créer **MC:INF**.
- FONCTION : Ajouter_Id_dans_MC:Inf
  Définition:
  Modifier **MC:INF** conformément à l'ajout d'une case de mémoire.
- FONCTION : Ajouter_TabId_dans_MC:Inf
  Définition:

Modifier **MC:INF** conformément à l'ajout d'un nombre donné de cases de mémoire.

- FONCTION : Mettre_Typ_dans_Id_MC:Inf
  Définition:
  Modifier **MC:INF** conformément à la modification du type d'une case de mémoire.
- FONCTION : Modifier_Val_dans_MC:Inf
  Définition:
  Modifier **MC:INF** conformément à la modification de la valeur d'une case de mémoire.
- FONCTION : Existe_Id_dans_MC:Inf
  Définition:
  Vérifier dans **MC:INF** l'existence d'une case d'un nom donné.
- FONCTION : Donner_Val_Id_MC:Inf
  Définition:
  Retourner la valeur d'une case de nom donné de **MC:INF.**

Sur OBJET **EC** :

- FONCTION : Creer_EC_vide
  Définition:
  Créer **EC.**
- FONCTION : Lire_Val_via_EC
  Définition:
  Lire une donnée.
- FONCTION : Ecrire_Val_sur_EC
  Définition:
  Ecrire une donnée.

Sur l'OBJET **ImageMC** :

- FONCTION : Creer_ImageMC
  Définition:
  Créer **ImageMC.**
- FONCTION : Copier_MC:Op_dans_ImageMC
  Définition:

Modifier **ImageMC** conformément aux déplacements de la *section visible* dans **MC:OP.**

- FONCTION : Copier_MC:Inf_dans_ImageMC
  Définition:
  Modifier **ImageMC** conformément aux déplacements de la *section visible* dans **MC:INF.**
- FONCTION : Montrer_ImageMC
  Définition:
  Afficher **ImageMc** sur l'écran "physique".

Sur l'OBJET **ImageEC** :
- FONCTION : Creer_ImageEC
  Définition:
  Créer **ImageEC.**
- FONCTION : Copier_EC_dans_ImageEC
  Définition:
  Copier **EC** dans **ImageEC.**
- FONCTION : Montrer_ImageEC
  Définition:
  Afficher **ImageMc** sur l'écran "physique".

Sur l'OBJET ***Symbole*** :
- FONCTION : Scanner_symbole
  Définition:
  Faire l'analyse lexicale d'un symbole.
- FONCTION : Accepter_symbole
  Définition:
  Faire l'analyse syntaxique d'un symbole.

2.6 L'architecture logique.

2.6.1 Pourquoi une architecture logique?

Dans le premier chapitre, nous avons vu que la construction de la marche à suivre faisait suite à l'énoncé du

problème. Lorsque le problème devient un *projet informatique*, la construction de la marche à suivre passe tout d'abord par l'établissement d'une *architecture logique* [D/VA87] [D/ME88], qui consiste à structurer le système à développer. Cette démarche est indispensable si l'on veut adopter une découpe modulaire du travail.

La première démarche de l'architecture logique consiste à structurer le système de manière hiérarchique. Il existe différentes façon d'établir la hiérarchisation. Celle qui s'adapte le mieux à une hiérarchisation de conception est la "UTILISE" basée sur la relation A "UTILISE" B qui implique que la validité de A dépend de la disponibilité d'une version correcte de B.

Voici la hiérarchisation que nous proposons pour l'outil **PROGRAIS**.

2.6.2 Hiérarchisation "UTILISE" en cinq niveaux.

2.6.2.1 *Niveau 5 : Le module fonctionnel.*

Le niveau de degré le plus élevé contient le "moteur" de l'outil **PROGRAIS**. Il est constitué d'un seul module. C'est le *module fonctionnel* de la figure suivante (*Fig. 2.4*):

Fig. 2.4 Le niveau 5

Ce module a pour objectif la prise en charge du bon fonctionnement de l'outil. Pour ce faire, il "UTILISE" les modules répartis dans les niveaux inférieurs.

2.6.2.2 *Niveau 4 : Un noyau fonctionnel en 3 sous-niveaux.*

Au niveau 4, sont regroupés trois modules qui se présentent comme un *noyau fonctionnel*. Ce sont les modules qui gèrent l'*interprétation* (au sens informatique) des algorithmes à exécuter. Ces trois modules forment un tout par rapport à la hiérarchie "UTILISE" de base. Ils sont, eux-mêmes, structurés

entre eux suivant une sous-hiérarchie "UTILISE". Cette sous-hiérarchie doit suivre un schéma classique d'un système du type Compilateur-Interpréteur[10]. Cela donne (*Fig. 2.5*):

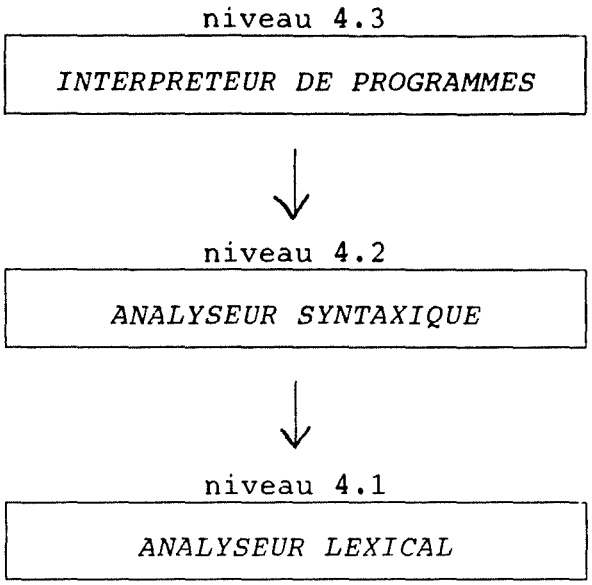


Fig. 2.5 Le niveau 4 et sa sous-hiérarchie "UTILISE"

Les trois modules de ce niveau sont évidemment indépendants de la fonctionnalité propre à l'outil **PROGRAIS**. Ils seront conçus de telle façon qu'ils puissent être utilisés par n'importe quel système informatique nécessitant un *interpréteur* de *langage de programmation*.

Le module de niveau 4.3, à savoir l'*INTERPRETEUR DE PROGRAMMES*, joue le rôle de *module fonctionnel* dans la sous-hiérarchie "UTILISE". C'est le seul module utilisé par le module *EXECUTEUR DE PROGRAMME* du niveau 5. Les deux autres modules sont donc invisibles pour le module de niveau 5.

2.6.2.3 *Niveau 3 : Quatre modules de gestion avancée.*

Le niveau suivant est formé de tous les modules de gestion avancée. Ils sont tous de même niveau par rapport à la hiérarchie "UTILISE".

---

(10) C'est principalement à partir du livre de Aho et al.[E/AH86] que nous avons établi la découpe de ce module.

Ces modules sont respectivement:

- un module s'occupant de la gestion des tables *internes* de la MEMOIRE CENTRALE, appelé

GERANTS DES TABLES INTERNES,

- un module s'occupant de tous les problèmes liés à l'édition de la partie visible de la MEMOIRE CENTRALE, appelé

EDITEUR DE MEMOIRE CENTRALE,

- un module s'occupant de tous les problèmes liés à la gestion et à l'édition de l'ECRAN du **Modèle**, appelé

GERANT D'ECRAN,

ainsi que,

- un module chargé de s'occuper de la gestion de tous les messages de dialogues tels que dialogues en cas d'erreur, dialogue de commentaires, dialogues d'aide, dialogue de fin d'exécution ..., appelé

GERANT DES MESSAGES.

Ce module est un module d'aide et ne dépend pas directement des spécifications fonctionnelles. Il sert à intégrer l'outil dans son contexte d'enseignement assisté par ordinateur.

Cela donne la figure suivante (*Fig. 2.6*) pour le niveau 3:

Fig. 2.6 Le niveau 3

2.6.2.4 *Niveau 2 : Un module de contrôle.*

Le niveau 2 comprend un module responsable du contrôle de la synchronisation des différentes éditions sur l'écran " physique", c-à-d sur l'écran en tant que *support de représentation.* C'est le module qui s'occupe de la gestion du multi-fenêtrage. C'est pourquoi, on l'appelle le

CONTROLEUR DE COMMUNICATION.

On a donc pour le niveau 2 (*Fig. 2.7*):

Fig. 2.7 Le niveau 2

2.6.2.5 *Niveau 1' et 1 : Deux modules utilitaires.*

Enfin, l'architecture se termine par deux modules utilitaires répartis en deux niveaux par rapport à la relation "UTILISE" globale. Le premier module de niveau 1' rassemble les primitives de base relatives au fenêtrage (création de fenêtre, ouverture de fenêtre, fermeture de fenêtre, ...) tandis que le second module de niveau 1 offrent un certain nombre de primitives de plus bas niveau (écriture rapide, temps d'attente, son, lecture de touche au clavier, ...). Ces deux modules sont représentés dans la figure suivante (*Fig. 2.6*):

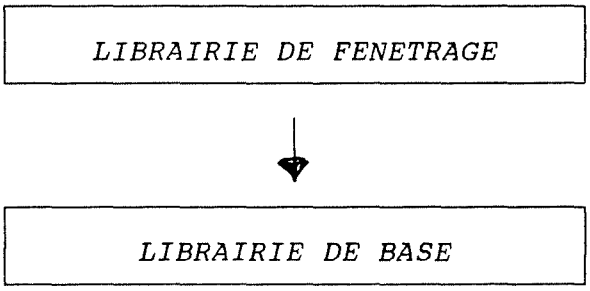


Fig. 2.8 Les niveaux 1' et 1

2.6.2.6 *Graphe de l'architecture logique.*

L'ensemble des niveaux ainsi dégagés sont assemblés en hiérarchie "UTILISE" suivant le graphe donné à la page suivante (*Fig. 2.9*).

2.6.3 Vers une structuration modulaire.

Ayant établit une structure hiérarchique pour le système, il est alors possible de concevoir la structuration modulaire qui n'est pas décrite ici mais qui peut être perçue en analysant le codage contenu dans l'annexe 2.

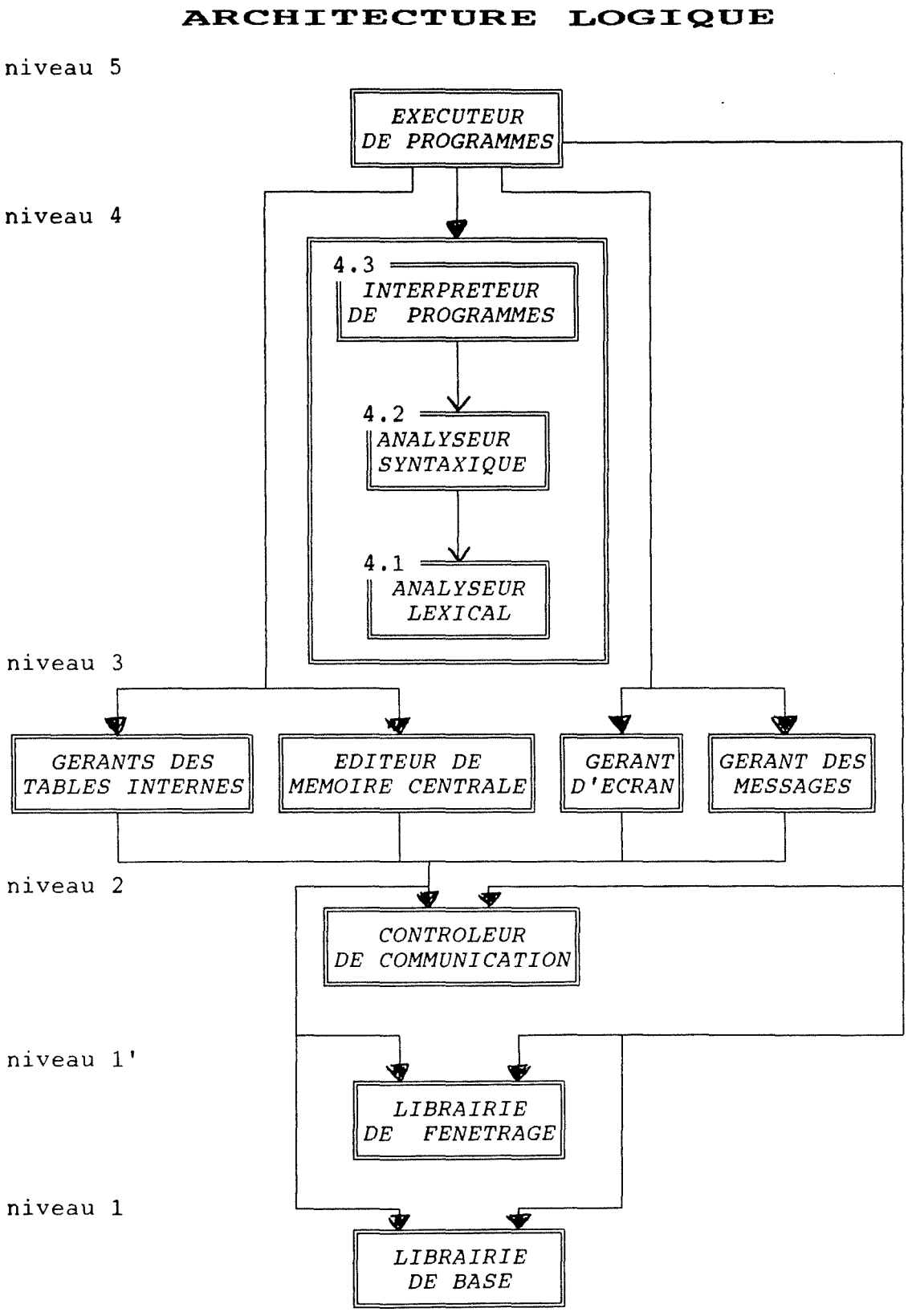


Fig. 2.9 Graphe de l'architecture logique de l'outil ***PROGRAIS***

2.7 Conclusion.

Dans ce deuxième chapitre, nous avons essayer d'indiquer la similarité qui existe entre la conception d'un problème élémentaire et la conception d'un projet informatique. Les bases sont posées pour réaliser un outil satisfaisant. Le résultat de toute la suite du développement se trouve contenu dans l'annexe 2 sous forme du codage. On peut y voir les choix relatifs à l'architecture physique ainsi que la démarche de développement par incréments successifs qui a l'avantage de montrer la correction des différents modules. La version finale actuelle de l'outil fonctionne. cette version ne satisfait pas encore à tous les objectifs mais elle constitue un prototype valable.

2.8 Sources bibliographiques.

Le chapitre 2 trouve ses sources dans bon nombre de lectures et dans plusieurs cours enseignées en licence et maîtrise en Informatique référencés dans la **BIBLIOGRAPHIE** sous les intitulés

C. Interactivité de l'édition.

D. Méthodologie de développement de logiciels.

E. Algorithmique et langages.

et

F. Théorie des programmes.

Calcul digital (*Bède, De temporum ratione, copié en Italie au début du XIV^e^ siècle*)

Chapitre 3

DEFINITION DU MINI LANGAGE DE PROGRAMMATION RECONNU

3.1 Introduction.

Dans le deuxième chapitre, nous avons explicité les caractéristiques générales pour le développement de l'outil informatique **PROGRAIS**. Nous avons ainsi proposé d'isoler la partie purement interprétation de langage (au sens informatique) dans un noyau fonctionnel, appelé *INTERPRETEUR DE PROGRAMMES*, indépendant des particularités propres à l'outil.

Selon le Petit Robert [I/RO67], le verbe *interpréter* signifie :

> *1° expliquer, rendre clair (ce qui est obscur),*
> *2° donner un sens à (qqch.), tirer une signification de.*

Ces définitions prennent toutes leurs significations dans le cas présent. En effet, c'est l'*INTERPRETEUR DE PROGRAMMES* qui rend clair et qui donne un sens aux programmes ou algorithmes à exécuter. C'est grâce à lui que l'utilisateur - ici un étudiant - peut communiquer ou dialoguer avec la machine ordinateur.

Pour que le dialogue soit possible, il faut que le langage dans lequel les programmes ou algorithmes sont représentés soit un langage de programmation capable d'être compris par l'*INTERPRETEUR DE PROGRAMMES*. Il est donc essentiel de définir

ce langage de façon claire, précise et exempte d'ambiguïtés. Cette définition constitue certainement une *étape fondamentale* dans le développement de l'outil **PROGRAIS**. Elle constitue l'objectif de ce troisième chapitre.

Pour ce faire, nous commencerons par expliciter ce qui nous a guidée dans le choix du langage de programmation retenu. Subséquemment, nous spécifierons les différents concepts de base de ce langage. Nous définirons ensuite la syntaxe abstraite qui découle de ces spécifications. Enfin, nous verrons comment passer aux représentations explicites de ce langage, appelées aussi syntaxes concrètes. Ce sont ces syntaxes concrètes qui pourront être intégrées dans l'outil **PROGRAIS**. Nous nous attarderons plus particulièrement à la définition de la syntaxe concrète de convention *Pascal* introduite dans la version actuelle. Nous exposerons aussi la définition d'une syntaxe concrète d'un *langage formel* aussi proche que possible du langage formel adopté par Lesuisse et Borsu [A/LE87].

3.2 Exigences de départ.

Le *langage de programmation* capable d'être interprété par l'outil **PROGRAIS** consiste en un mini langage limité à un certain nombre de concepts. L'identification des concepts retenus a été guidée par les deux exigences suivantes:

1) définir un *langage de programmation* permettant d'utiliser la *Méthode d'Initiation aux Raisonnements de la Programmation de Lesuisse et Borsu* [A/LE87] pour l'acquisition des principes fondamentaux de la programmation,

2) définir un *langage de programmation* capable d'être pris en compte dans une version informatisée de leur **Modèle Général de l'Ordinateur** décrit dans le chapitre 1.

Il faut donc que le mini langage reconnu par l'outil **PROGRAIS** soit suffisamment fourni pour permettre d'illustrer

les principaux concepts sur lesquels repose la programmation tout en n'étant pas trop évolué pour pouvoir être incorporé dans le cadre du modèle.

Si le mini langage de programmation retenu respecte ces deux exigences, alors l'outil informatique **PROGRAIS** sera à même de traiter des problèmes intéressants et, par ce fait, il pourra remplir de manière satisfaisante le rôle pédagogique souhaité.

3.3 Spécifications des concepts de base.

3.3.1 Présentation des concepts.

La définition d'un langage de programmation fait intervenir un certain nombre de concepts de base. Il faut tout d'abord spécifier les types de données qui donnent le domaine de valeurs que peuvent prendre les objets manipulés. Il s'agit également de préciser la forme que peuvent prendre les objets. Le concept suivant est le concept d'opération permettant de mettre-en-oeuvre les différents modes de raisonnements. Enfin, il convient de préciser clairement ce qu'on entend par programme ou algorithme.

3.3.2 Types de données.

Les deux types de données retenus sont:

le type **entier** et le type **booléen.**

Ces deux types de données définissent le *domaine de valeurs* que peuvent prendre les objets manipulés. Comme indiqué dans le Chapitre 1, ces objets sont soit des informations émanant de l'utilisateur soit des résultats obtenus au cours d'une exécution(1).

Dans le mini langage, les informations et résultats pris en compte peuvent être respectivement:

---

(1) Cf. la section 1.4.1.1 du Chapitre 1.

- des *constantes* ou constructions qui représentent en elles-même une valeur,

  . dans le cas d'une constante de type **entier**, il s'agit d'un nombre éventuellement affecté d'un signe + ou - ,

  . dans le cas d'une constante de type **booléen**, il s'agit d'une valeur représentée par un *nom* qui peut différer en fonction du langage, comme

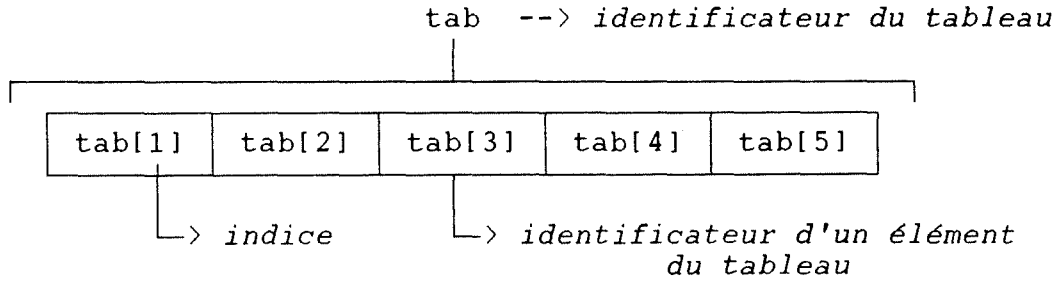
  vrai faux true false 0 1

- des *variables simples* identifiables par un *nom*, appelé *identificateur* de la variable,

- des *expressions* formées de variables simples, de constantes et d'opérateurs opérant sur les valeurs déjà définies pour produire de nouvelles valeurs,

ainsi que:

- des *variables "tableau"* identifiables par un *nom* unique, appelé *identificateur du tableau* et composées d'un nombre prédéterminé de variables simples de même type repérables grâce à un *indice* numérique joint au nom , comme illustré sur cet exemple :



Les *opérateurs* intervenant dans les expressions sont les opérateurs classiques unaires et binaires. Le langage devrait au moins admettre:

. les *opérateurs arithmétiques* d'addition, de soustraction, d'inversion de signe, de multiplication et de division(2),

. l' *opérateurs booléen* de négation.

Il pourrait aussi admettre :

. les *opérateurs relationnels* d'égalité, d'inégalité, d'ordres stricts et non stricts.

Il est à noter que dans leur ouvrage [A/LE87], Lesuisse et Borsu définissent comme types de données le type **entier**, le type **réel** et le type **caractère**. Ensuite, dans le cours de leur méthode, ils se servent des types **entier** et **réel**. Le fait de ne pas retenir le type **réel** dans le mini langage de programmation de l'outil informatique **PROGRAIS** est un *choix délibéré*. En effet, les différentes représentations des réels, telles que représentation à virgule fixe, représentation à virgule flottante ou même représentation scientifique, sont souvent source de confusion chez les débutants en informatique. De plus, la visualisation de base sur l'écran de l'ordinateur dans les compilateurs classiques ne correspond pas à la représentation généralement utilisée en mathématique. Nous pensons que dans un tout premier stade de l'apprentissage de la programmation, la perception des différents raisonnements n'est sûrement pas diminuée en se limitant au type **entier**. Or, c'est à ce premier stade de l'apprentissage que l'outil **PROGRAIS** intervient. Cependant, si le module *INTERPRETEUR DE LANGAGE* est bien conçu, l'annexion du type **réel** dans le langage ne devrait présenter aucun problème en soi! La difficulté se situerait uniquement au niveau des conventions à adopter pour la visualisation des nombres réels sur l'ECRAN.

Si le type **booléen** a également été retenu, c'est parce qu'il est nécessaire pour construire l'outil. Ceci permet

(2) Pour le type **entier**, il y a deux opérateurs de division, l'opérateur donnant la partie entière de la division et l'opérateur donnant le reste de la division.

aussi de montrer comment travailler avec plusieurs types de données, ce qui présente déjà un intérêt en soi.

3.3.3 Opérations.

Les opérations reconnues sont les opérations classiques de la programmation associés aux différents modes de raisonnements fondamentaux décrits dans la *Méthode d'Initiation aux Raisonnements de la Programmation de Lesuisse et Borsu* [A/LE87]. On peut les classer en opérations simples et en opérations structurées qui font elles-même appel à d'autres opérations.

Les opérations *simples* sont:

l'opération vide, l'opération d'affectation, l'opération de lecture et **l'opération d'écriture**

Les opérations *structurées* sont:

L'opération composée, l'opération conditionnelle et **l'opération répétitive.**

Toutes ces opérations peuvent être décrites succinctement de la façon suivante:

Opération vide : Opération qui consiste à ne rien faire.

Opération d'affectation : Opération interne qui consiste à affecter une nouvelle valeur à une variable (ou élément d'une variable dans le cas d'une variable tableau); la valeur sera soit une constante, soit la valeur connue d'une autre variable, soit le résultat d'une expression.

Opération de lecture : Opération qui consiste à affecter à une variable (ou élément d'une variable dans le cas d'une variable tableau) une valeur transmise au clavier par l'utilisateur; c'est, en fait, une opération d'affectation externe complémentaire de l'opération d'affectation interne précédente.

Opération d'écriture : Opération qui consiste à communiquer à l'utilisateur via l'écran soit une constante, soit la valeur résultat d'une expression, soit encore la valeur connue d'une variable (ou élément d'une variable dans le cas d'une variable tableau).

Opération composée : Opération qui consiste en une suite d'autres opérations pouvant être des opérations elles-même composées.

Opération conditionnelle : Opération dont la réalisation dépend de la vérification d'une condition, la condition étant une expression à valeur booléenne.

Opération répétitive : Opération qui consiste à reproduire un certain nombre de fois une autre opération, la détermination du nombre de fois dépendant du résultat d'une condition suivant le schéma *"tant que* la condition est vérifiée, *répéter* l'opération".

Si le mini langage proposé permet d'illustrer la plupart des modes de raisonnements fondamentaux décrits dans la méthode de Lesuisse et Borsu [A/LE87], il n'autorise cependant pas le mode de raisonnement plus complexe qui consiste à faire appel à un raisonnement déjà défini et auquel est associée l'opération dite **opération de procédure.** Cette opération n'a pas été retenue parce qu'il nous a semblé trop difficile de l'incorporer dans le traitement informatisé du **Modèle général de l'Ordinateur** dans l'outil **PROGRAIS.** Notons cependant que, à strictement parlé, l'**opération de lecture** et l'**opération d'écriture** sont en fait des cas particuliers d'opérations de procédure. Elles sont d'ailleurs définies comme telles dans les langages standards. Ce qui veut dire que, malgré son caractère limité, le langage retenu permet indirectement d'illustrer le mode de raisonnement d'appel de procédure.

Le fait que la condition à vérifier dans l'opération conditionnelle soit une expression booléenne justifie la prise en compte du type **booléen** dans le mini langage de programmation retenu. En effet, le langage qui est défini dans

ce chapitre ne sert pas uniquement à représenter les programmes ou algorithmes qui sont exécutés dans le **Modèle Général de l'Ordinateur**. C'est ce même langage qui est incorporé dans le module *INTERPRETEUR DE PROGRAMMES*.

Pour l'opération répétitive, seul un schéma a été retenu. Ce n'est évidemment pas le seul schéma de répétition, mais il est cependant admis que ce schéma est le meilleur si on n'en retient qu'un. L'ajout d'autres schémas dans la définition de l'opération répétitive ne devrait pas en principe poser de problèmes majeur.

3.3.4 Programme ou algorithme.

Le concept fondamental d'un langage de programmation et, par conséquent, du mini langage de programmation retenu pour l'outil **PROGRAIS** est le concept de

programme ou **algorithme.**

Ce concept peut être défini de façon plus ou moins détaillée. Dans le cas présent, il sera défini ainsi:

> Un **programme** ou **algorithme** est une *suite d'opérations* reconnues s'appliquant sur des *données* d'un *type* reconnu.

La définition donnée est délibérément assez vague laissant ainsi le plus de liberté possible pour la définition d'un *langage formel* d'exécution de programmes tel qu'il est utilisé dans la méthode de Lesuisse et Borsu [A/LE87]. Dans cette définition, rien n'est précisé quant à la manière dont les objets manipulés et leurs types sont déclarés En d'autres mots, cette définition ne donne aucune indication sur la façon dont les cases de la partie "INFORMATIONS" de la MEMOIRE CENTRALE sont réservées. Cela permet de reporter le choix entre des déclarations anticipées (c.-à-d. réservation avant exécution) ou non (c.-à-d. réservation en cours d'exécution) au stade de la définition explicite ou *concrète* de la représentation du mini langage de programmation.

3.4 Définition de la syntaxe abstraite.

Une des conditions pour que l'outil **PROGRAIS** soit un bon outil[3] consiste à prévoir la possibilité d'admettre des algorithmes rédigés dans divers langages de programmation. Il est donc essentiel de concevoir l'outil **PROGRAIS** de telle façon qu'il soit le moins possible lié à une représentation particulière du mini langage dont les concepts viennent d'être spécifiés. Pour cela, il faut avant tout s'attacher à dégager la sémantique du langage avant d'analyser les détails d'écriture de la représentation de ce langage. L'écriture dans une représentation particulière devient alors essentiellement un problème de traduction.

La syntaxe qui convient pour dégager la sémantique d'un langage de programmation est appelée *syntaxe abstraite*. La section précédente consistait en une ébauche de syntaxe abstraite exprimée en français. Mais la langue française ne permet pas une formulation à la fois précise et concise. Ceci est possible en utilisant un méta-langage appelé *langage de définition de syntaxe abstraite*. La syntaxe abstraite se présente alors comme l'ensemble des symboles appelés *types syntaxiques* et de leurs règles de production appelées *règles syntaxiques*.

La syntaxe abstraite du mini langage de programmation retenu pour l'outil **PROGRAIS** est présentée dans la *Table 3.1*. Le langage de définition de syntaxe abstraite utilisé est un langage conforme à celui qui est proposé par Le Charlier dans son cours de Théorie des programmes [F/LE88] ou par Tennent [F/TE81]. Les types syntaxiques sont représentés par une seule lettre et les règles syntaxiques sont exprimées en utilisant la forme BNF (Backus-Naur Form). Les symboles suivants sont des métasymboles de cette forme BNF :

::= signifie "est défini comme"
( ) entoure les éléments facultatifs
* (astérisque en exposant) indique les éléments à répéter

(3) Il s'agit de la quatrième condition donnée à la section 2.1.2 du chapitre 2.

¦ (barre verticale) signifie "ou"

Tous les autres symboles et les mots en caractère gras font partie intégrante du vocabulaire propre au langage. Ce sont les symboles spéciaux et les mots réservés du langage. Les conventions linguistiques ont été prises en conformité avec le *langage formel* défini dans la méthode de Lesuisse et Borsu [A/LE87].

Cela donne:

Types Syntaxiques

V ∈ Const.	{ Constantes	}
I ∈ Id.	{ Identificateurs simples	}
A ∈ Tab.	{ Identificateurs de tableau	}
E ∈ Expr.	{ Expressions	}
O ∈ Op.	{ Opérations	}
D ∈ Decl.	{ Déclarations	}
P ∈ Prog.	{ Programmes	}

Règles Syntaxiques

P ::= (D*) O
D ::= I : T ¦ A : **tableau de** T
T ::= **entier** ¦ **booléen**
O ::= **vide** ¦ I -> E ¦ A[E] -> E
¦ **lire** I ¦ **afficher** E
¦ O* ¦ **si** E **alors** O **sinon** O
¦ **tant que** E **répéter** O
E ::= V ¦ I ¦ A[E] ¦ UE ¦ EBE
U ::= + ¦ - ¦ **non**
B ::= + ¦ - ¦ * ¦ **div** ¦ **mod**(4)
¦ = ¦ <> ¦ < ¦ ≤ ¦ > ¦ ≥

Table 3.1 Syntaxe Abstraite du langage reconnu par l'outil PROGRAIS

---

(4) Les opérateurs **div** et **mod** sont les deux opérateurs de division pour les types entiers et booléens. L'opérateur de division / ne peut être retenu parce qu'il donne un résultat de type réel.

Conjointement à la syntaxe abstraite, il convient de définir les domaines sémantiques des valeurs de base du langage. Avec comme types de données, le type **entier** et le type **booléen**, ces domaines sont (*Table 3.3*):

V = **B** U **Z** (union de **B** et de **Z**)

où

B = { vrai, faux }

c.-à-d. l'ensemble des valeurs booléennes

Z = {...,-2,-1,0,1,2,...}

c.-à-d. l'ensemble des valeurs entières

Table 3.2 Domaines sémantiques des valeurs de base du langage reconnu par l'outil PROGRAIS

Par domaine sémantique, on entend le domaine abstrait des valeurs, non lié à une notation quelconque de cette valeur. Ainsi, à ce stade-ci,

106 ou 1.06E+2 ou 1101010B ou CVI ou cent six

ne sont que des notations ou représentations différentes pour un seul objet abstrait, à savoir l'**entier 106** du domaine sémantique **Z**.

3.5 De la syntaxe abstraite aux syntaxes concrètes.

La syntaxe abstraite du langage reconnu par l'outil informatique **PROGRAIS** définit les structures syntaxiques disponibles dans le langage mais elle ne permet pas de conclure si le texte explicite du programme ou algorithme est bien formé. Pour cela, il faut faire appel à une syntaxe dite syntaxe concrète. C'est cette syntaxe qui doit être précisée à l'utilisateur de l'outil informatique **PROGRAIS.** C'est également cette syntaxe qui sera effectivement prise en compte dans le sous-module *ANALYSEUR SYNTAXIQUE* du noyau fonctionnel *INTERPRETEUR DE PROGRAMME* de l'outil.

Pour chaque représentation de langage de programmation, il suffit de définir la *syntaxe concrète* adéquate. Dans chaque cas, les différents constituants à préciser sont respectivement les symboles de bases du langage, les mots réservés, les identificateurs standards, les délimiteurs et, enfin et surtout, l'ensemble des règles syntaxiques.

Pour la réalisation de la version actuelle de l'outil **PROGRAIS**, nous avons défini une *syntaxe concrète* de convention *Pascal* appelée *langage DEMO PASCAL* entièrement compatible avec la syntaxe concrète du *langage Pascal* standard [G/JE78] ou même du *langage Turbo Pascal* de Borland [G/BOxx]. En conséquence, l'outil informatique **PROGRAIS** doit pouvoir exécuter n'importe quel programme ou algorithme écrit dans un de ces langages tant qu'il ne contient que des structures syntaxiques définies dans notre syntaxe abstraite. Pour montrer l'indépendance de l'outil par rapport au choix de représentation, nous donnons également la définition d'une syntaxe concrète représentant un *langage formel* aussi proche que possible du langage formel adopté dans la méthode de Lesuisse et Borsu [A/LE87] et qui peut être incorporée dans l'outil **PROGRAIS**.

3.6 Définition d'une syntaxe concrète de convention *Pascal*.

3.6.1 Les symboles de bases.

Les symboles de base de la syntaxe concrète de convention *Pascal*, appelée *langage DEMO PASCAL* sont les lettres, les chiffres et certains symboles spéciaux.

Lettres :

les 26 lettres de l'alphabet en majuscule et en minuscule ainsi que le symbole souligné _
(aucune distinction n'est faite entre les majuscules et les minuscules)

Chiffres :

0 1 2 3 4 5 6 7 8 9

Symboles spéciaux :

+ - * = < > ( ) [ ] . , ; :

Certains opérateurs et délimiteurs sont construits avec deux symboles spéciaux. Ce sont:

l'opérateur d'affectation :=

les opérateurs relationnels <> <= >=

le délimiteur d'intervalle ..

3.6.2 Les Mots Réservés.

Les mots réservés sont des constructions de symboles de base qui font partie intégrante du langage. Ils ne peuvent pas être redéfinis et, par conséquent, ils ne peuvent pas être utilisés par l'utilisateur comme identificateur pour une variable. Dans le *langage DEMO PASCAL*, ils sont au nombre de 16, à savoir les mots (par ordre alphabétique):

array	**begin**	**div**	**do**
else	**end**	**if**	**mod**
not	**of**	**program**	**read**
then	**var**	**while**	**write**

3.6.3 Les Identificateurs standards.

Le *langage DEMO PASCAL* a de plus quatre identificateurs standards prédéfinis qu'il est préférable de ne pas redéfinir(5):

---

(5) Les identificateurs standards ne sont pas considérés comme des mots réservés afin de se conformer au *langage Pascal* standard. La seule différence par rapport au *langage Pascal standard* concernent les mots **read** et **write**. Dans le *langage DEMO PASCAL*, ils sont considérés comme étant des mots réservés et non comme des identificateurs standards. En effet, les opérations de lecture et d'écriture sont définies comme étant des opérations de base du langage et non comme des opérations de procédure.

identificateurs de Types : **Boolean** **integer**

identificateurs de constantes de type booléen : **false** **true**

3.6.4 Les Délimiteurs.

Les délimiteurs sont soit des séparateurs, soit des parenthèses. Pour le *langage DEMO PASCAL*

les séparateurs sont :

. , ; : ..
ainsi qu'un blanc ou une fin de ligne

les parenthèses sont :

begin **end** ( ) [ ]

3.6.5 Les Opérateurs.

Les opérateurs interviennent dans les expressions. Dan le *langage DEMO PASCAL,* les opérateurs retenus sont les suivants:

pour les opérateurs unaires :

l'opérateur de négation booléenne (negop) : **not**

les opérateurs arithmétiques de signe (signop) : + -

pour les opérateurs binaires :

les opérateurs arithmétiques d'addition (addop) : + -

les opérateurs arithmétique de multiplication (mulop) :
*** div mod**

les opérateurs de relation (relop) : = <> < <= > >=

Ces opérateurs ont été classés en cinq catégories en fonction de leurs niveaux de priorité lors du traitement des expressions.

3.6.6 Les Règles Syntaxiques.

Il ne reste plus qu'à énoncer les règles syntaxiques du *langage DEMO PASCAL*. Ces règles résument tout ce qu'il faut connaître pour écrire une représentation correcte d'un programme ou algorithme en *langage DEMO PASCAL* dans la *Table 3.3*. Comme pour la syntaxe abstraite, le méta-langage utilisé est la forme BNF mais, cette fois, dans sa forme étendue dite forme EBNF (Extended Backus-Naur Form) qui consiste à utiliser des règles de production récursives pour éviter l'usage d'un nombre arbitraire de répétitions.
Par exemple :

dans la forme EBNF, la règle

entss := chiffre (chiffre)*

qui définit un nombre entier sans signe est remplacée par la règle récursive

entss := chiffre | entss chiffre

Ce sont les règles de la *Table 3.3* qui ont utilisées pour la réalisation du sous-module *ANALYSEUR SYNTAXIQUE* de l'*INTERPRETEUR DE PROGRAMMES*. Les règles y sont numérotées pour faciliter et systématiser le travail d'intégration.

```
1  programme ::= en_tete bloc .

2  en_tete ::= program id ;
3  bloc ::= partie_declarations partie_opérations

4  partie_declarations ::= ϵ | var liste_decl_variables
5  liste_decl_variables ::= decl_variables
   | liste_decl_variables
6  decl_variables ::= liste_id : decl_type ;
7  liste_id ::= id | liste_id , id
8  decl_type ::= type_simple | type_tableau
```

Table 3.3 Syntaxe Concrète du langage DEMO PASCAL

```
9  type_tableau ::= array[entss .. entss] of type_simple
10 type_simple ::= integer | boolean

11 partie_operations ::= operation_composee
12 operation_composee ::= begin liste_operation end
13 liste_operation ::= operation | liste_operation ; operation
14 operation ::= operation_simple | operation_structuree
15 operation_simple ::= operation_vide | operation_affectation
   | operation_lecture | operation_ecriture
16 operation_structuree ::= operation_composee
   | operation_conditionnelle | operation_repetitive
17 operation_vide ::= ϵ
18 operation_affectation ::= variable := expression
19 operation lecture ::= read( liste_variable )
20 operation_ecriture ::= write( liste_expression )
21 operation_conditionnelle ::= if condition then operation
   | if condition then operation else operation
22 operation_repetitive ::= while condition do operation

23 liste_variable ::= variable | liste_variable , variable
24 liste_expression ::= expression | liste_expression ,
   expression
25 variable ::= id | id[expression]

26 condition ::= expression | expression relop expression
27 expression ::= terme | signop terme | expression addop
   terme
28 terme ::= facteur | terme mulop facteur
29 facteur ::= cstss | variable | negop facteur | (expression)

30 id ::= lettre (lettre_ou_chiffre)⁹
31 cst ::= true | false | nbrss | signop nbrss
32 cstss ::= true | false | nbrss
```

Table 3.3 Syntaxe Concrète du langage DEMO PASCAL (suite)

```
33 nbrss ::= entss
34 entss ::= chiffre ¦ entss chiffre

35 relop ::= = ¦ <> ¦ < ¦ <= ¦ > ¦ >=
36 signop ::= + ¦ -
37 addop ::= + ¦ -
38 mulop ::= * ¦ div ¦ mod
39 negop ::= not
40 ϵ ::=                              (vide)

41 lettre_ou_chiffre ::= lettre ¦ chiffre
42 lettre ::=
   a¦b¦c¦d¦e¦f¦g¦h¦i¦j¦k¦l¦m¦n¦o¦p¦q¦r¦s¦t¦u¦v¦w¦x¦y¦z¦
   A¦B¦C¦D¦E¦F¦G¦H¦I¦J¦K¦L¦M¦N¦O¦P¦Q¦R¦S¦T¦U¦V¦W¦X¦Y¦Z¦_
43 chiffre ::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9
```

Table 3.3 Syntaxe Concrète du langage DEMO PASCAL (fin)

3.6.7 Simplifications pour la version actuelle.

Dans la version actuelle de l'outil **PROGRAIS**, les règles de DEMO PASCAL ne sont pas encore toute opérationnelle. Reste encore à incorporé les détails pour l'opération conditionnelle, pour l'opération répétitive et pour tout ce qui concerne les expressions. A strictement parlé au niveau de la définition de la syntaxe concrète, c'est comme si certaines règles avaient été modifiées de la façon suivante:

```
21 operation_conditionnelle ::= ϵ
22 operation_repetitive ::= ϵ
```

```
26 condition ::= expression
27 expression ::= terme
28 terme ::= facteur
29 facteur ::= cstss ¦ id
```

Ceci revient à assimiler les opérations conditionnelle et répétitive à l'opération vide et à ramener une expression soit à un nombre entier positif soit à un identificateur de variable simple.

3.7 Définition d'une syntaxe concrète pour un langage formel.

3.7.1 Les symboles de bases.

Les symboles de base sont les même que ceux du *langage DEMO PASCAL*, aux deux exceptions suivantes:

dans ce cas, l'opérateur d'affectation sera noté:

l'opérateur d'affectation : ->

et le délimiteur d'intervalle n'est pas utilisé.

3.7.2 Les Mots Réservés.

Les mots réservés du *langage formel* sont au nombre de 12, à savoir les mots (par ordre alphabétique) :

afficher	**alors**	**div**	**finsi**
fintant_que	**lire**	**mod**	**non**
répéter	**si**	**sinon**	**tant_que**

3.7.3 Les Identificateurs standards.

Le *langage formel* a quatre identificateurs standards prédéfinis qu'il est préférable de ne pas redéfinir:

identificateurs de Types : **Booleen** **entier**

identificateurs de constantes de type booléen : **faux** **vrai**

3.7.4 Les Délimiteurs.

Les délimiteurs du *langage formel* sont

les séparateurs

. , ; :

ainsi qu'un blanc ou une fin de ligne

les parenthèses

( ) [ ]

3.7.5 Les Opérateurs.

Les opérateurs retenus sont les mêmes que pour le *langage DEMO PASCAL*. La notation est également la même sauf pour l'opérateur booléen de négation qui est à présent noté **non**.

3.7.6 Les Règles Syntaxiques.

Les règles syntaxiques proposées pour le *langage formel* sont contenues dans la *Table 3.4*.

```
1  programme ::= bloc .

2  bloc ::= partie_opérations

3  partie_operations ::= operation_composee
4  operation_composee ::= liste_operation
5  liste_operation ::= operation ¦ liste_operation ; operation
6  operation ::= operation_simple ¦ operation_structuree
7  operation_simple ::= operation_vide ¦ operation_affectation
   ¦ operation_lecture ¦ operation_ecriture
8  operation_structuree ::= operation_composee
   ¦ operation_conditionnelle ¦ operation_repetitive
9  operation_vide ::= ϵ
```

Table 3.4 Syntaxe Concrète du Langage Formel

```
10 operation_affectation ::= variable -> expression
11 operation lecture ::= lire liste_variable
12 operation_ecriture ::= afficher liste_expression
13 operation_conditionnelle ::= si condition alors operation
   finsi | si condition alors operation sinon operation finsi
14 operation_repetitive ::= tant_que condition répéter
   operation fintant_que
15 liste_variable ::= variable | liste_variable , variable
16 liste_expression ::= expression | liste_expression ,
   expression
17 variable ::= id | id[expression]

18 condition ::= expression | expression relop expression
19 expression ::= signop terme | expression addop terme
20 terme ::= facteur | terme mulop facteur
21 facteur ::= cstss | variable | negop facteur | (expression)

22 id ::= lettre (lettre_ou_chiffre)⁹
23 cst ::= vrai | faux | nbrss | signop nbrss
24 cstss ::= vrai | faux | nbrss
25 nbrss ::= entss
26 entss ::= chiffre | entss chiffre

27 relop ::= = | <> | < | <= | > | >=
28 signop ::= + | - | ϵ
29 addop ::= + | -
30 mulop ::= * | div | mod
31 negop ::= non

32 ϵ ::=                              (vide)

33 lettre_ou_chiffre ::= lettre | chiffre
```

Table 3.4 Syntaxe Concrète du Langage Formel (suite)

```
34 lettre ::=
   a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
   A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|_
35 chiffre ::= 0|1|2|3|4|5|6|7|8|9
```

Table 3.4 Syntaxe Concrète du langage formel (fin)

3.8 Conclusion.

De ce chapitre, nous retirons beaucoup d'enseignements sur les mécanismes de définition des langages de programmation et sur les difficultés rencontrées pour garder une certaine rigueur lorsqu'il s'agit de définir un langage de programmation.

Comme exemple des difficultés rencontrées, nous citerons le cas des *expressions*. Les définitions varient suivant les sources consultées et il ne nous a pas été facile de fixer une définition à la fois rigoureuse et adaptée au mini langage de l'outil **PROGRAIS**. Nous avons décidé de dissocier la notion de condition de celle d'expression tout comme proposé par Wirth dans la définition de son langage PL/0 contenu dans le chapitre 5 de son livre "Algorithms + Data Structures = Programs" [F/WI76].

3.9 Sources bibliographiques.

Pour le rédaction de ce chapitre 3, nous avons consulté et utilisé plusieurs travaux référencés dans la **BIBLIOGRAPHIE** sous l'intitulé

F. Théorie des Programmes.

Nous donnons également en référence sous l'intitulé

E. Algorithmes et Langages.

quelques travaux plus généraux qui nous avons trouvé très utile pour une première approche de l'étude des langages de programmations.

Astronome, copiste et computiste (*Psautier de Saint Louis, XIII^e^ siècle*)

Chapitre 4

DOCUMENTATION POUR L'OUTIL

4.1 Introduction.

En fin de phase de développement, on se trouve en présence d'un outil réalisé, ou du moins d'un produit qui peut fonctionner. Commence alors le dernière étape du cycle de vie de l'outil : son utilisation.

Une des qualités essentielles d'un bon outil informatique doit être, à notre sens, sa facilité d'emploi surtout lorsqu'il s'agit d'un outil destiné à l'enseignement. C'est ce que nous avons essayé de faire lors de la réalisation.

Une autre qualité essentielle consiste aussi à bien documenter le produit fini. Il y a différentes façons d'y arriver. Tout cela fait partie de la phase de mise au point de l'outil. On peut y arriver par des aides à l'écran tels ceux qui ont été prévus dans l'outil (et qui peuvent être améliorés) mais, aussi et surtout, par une bonne documentation d'accompagnement.

C'est cette documentation qui fait l'objet de ce dernier chapitre sous la forme d'un *guide de l'utilisateur* dans lequel on peut trouver le cadre général de l'outil, quelques notions du contexte dans lequel il s'intègre, le matériel qu'il requiert, comment le démarrer ainsi qu'un mode d'emploi détaillé.

4.2 Guide de l'utilisateur.

GUIDE DE L'UTILISATEUR

de

"PROGRAIS"

version 1.0

Outil Informatique annexe à la Méthode

d'*Initiation aux* RAI*Sonnements de la* PROG*rammation*

de R. Lesuisse et A. Borsu.

écrit par Anne de Baenst-Vandenbroucke

Qu'est-ce-que PROGRAIS ?

PROGRAIS est un outil informatique d'enseignement assisté par ordinateur (EAO) qui intervient dans la première phase de l'apprentissage de la Programmation. C'est un outil annexe à la *Méthode d'Initiation aux Raisonnements de la Programmation* de R. Lesuisse et A. Borsu[1] qui est enseignée dans le cours d'*Introduction à l'Informatique* délivré aux étudiants de Première Candidature en Sciences Economiques, Politiques et Sociales des Facultés Universitaires Notre-Dame de la Paix de Namur.

L'outil **PROGRAIS** est destiné à illustrer comment un ordinateur procède pour exécuter un programme (ou algorithme) en simulant l'exécution dans le cadre du **Modèle Général d'un Ordinateur** mis au point par les deux auteurs de la méthode précitée.

Qu'est-ce-que le Modèle Général d'un Ordinateur?

Le **Modèle Général d'un Ordinateur**[2] a été conçu par R. Lesuisse et A. Borsu pour leur *Méthode d'Initiation aux Raisonnements de la Programmation* dans le but d'asseoir le début de la phase d'apprentissage de la Programmation sur des

(1) Référence : Lesuisse R. et Borsu A., *Initiation aux Raisonnements de la Programmation*, Presses Universitaires de Namur, 1987 (en cours de réédition).

(2) On trouvera plus de détails sur ce sujet dans l'ouvrage de R. Lesuisse et A. Borsu déjà cité ainsi que dans le Chapitre 1 de ce mémoire.

bases solides et d'amener le plus rapidement et le plus efficacement les étudiants à concevoir de façon autonome des programmes "corrects".

Le **Modèle Général d'un Ordinateur** donne une représentation très schématique des principaux éléments matériels (ou "hardware") d'un ordinateur. Il n'a pas l'ambition de représenter dans tous les détails, un vrai ordinateur. Il est simplement destiné à faire comprendre comment un ordinateur travaille lorsqu'il exécute un programme (ou algorithme).

Les éléments matériels schématisés dans le **Modèle Général d'un Ordinateur** sont non seulement les éléments apparents mais également les éléments vitaux internes à la machine et donc normalement non observables. Dans la figure qui suit (*Fig. 1*), on peut voir comment se présente le **Modèle Général d'un Ordinateur** au cours de l'exécution d'un programme.

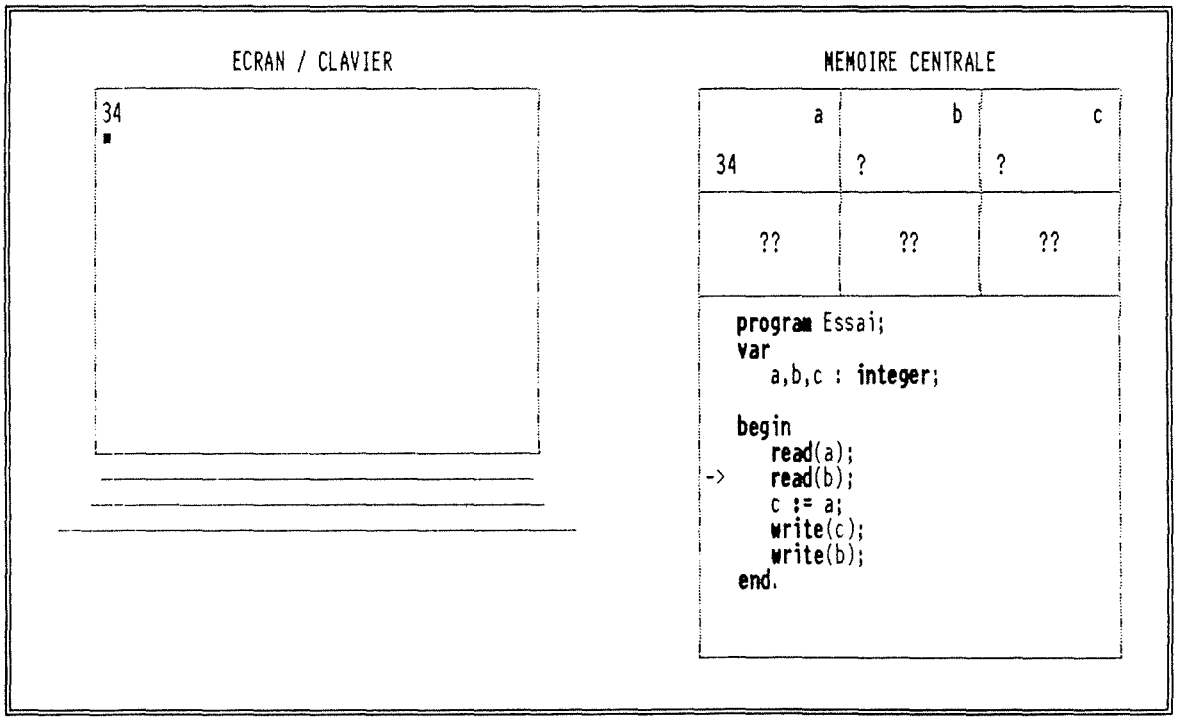


Fig. 1 Modèle Général de l'Ordinateur
- exemple avec un programme représenté en langage Pascal -
(Situation au cours de la deuxième opération de lecture)

Le programme pris en exemple(3) est représenté en *langage Pascal*. La figure montre la situation au cours de la deuxième opération de lecture du programme.

Ainsi qu'on peut le constater:

- Le dispositif de communication ECRAN/CLAVIER est représenté tel qu'il apparaît en réalité.

- L'élément MEMOIRE CENTRALE de l'UNITE CENTRALE de l'ordinateur est schématisé sous la forme d'un casier à deux parties avec
  . une partie formée de cases réservées à l'enregistrement des informations émanant de l'utilisateur ainsi qu'aux résultats des opérations exécutées,
  et
  . une partie réservée à l'enregistrement des lignes du texte du programme (ou algorithme).

- Il est plus difficile de représenter les divers circuits imprimés qui composent l'élément PROCESSEUR CENTRAL de l'UNITE CENTRALE. Les circuits gérant l'ordre d'exécution des opérations figurant dans le texte du programme sont schématisés sous la forme du signet "->" placé dans la partie de la MEMOIRE CENTRALE réservée au texte du programme face à l'opération à exécuter ou en cours d'exécution. Les circuits gérant l'exécution proprement dite des opérations ne sont pas représentés de manière explicite, mais il est cependant possible d'observer comment ils opèrent au vu des modifications intervenant dans les cases de la MEMOIRE CENTRALE du modèle au cours de l'exécution.

---

(3) Il s'agit du programme ESS4.PAS fourni avec **PROGRAIS**.

Le Modèle Général d'on Ordinateur dans PROGRAIS

Pour reproduire le **Modèle Général d'un Ordinateur** sur l'écran de l'ordinateur, **PROGRAIS** utilise deux fenêtres qui se superposent: une, plein écran, pour représenter l'ECRAN du modèle et une autre qui vient se superposer sur la moitié droite pour représenter la MEMOIRE CENTRALE comme montré sur la figure qui suit (*Fig. 2*):

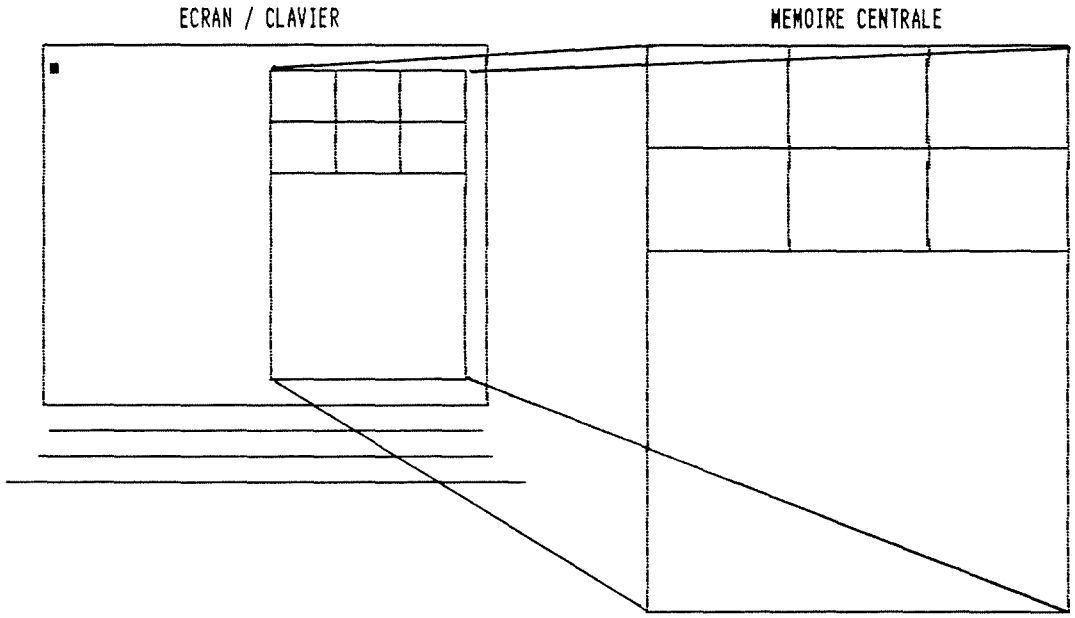


Fig. 2 Représentation du Modèle Général dans PROGRAIS

Il est clair que, dans cette figure, la fenêtre représentant la MEMOIRE CENTRALE est limitée en grandeur et qu'il n'est donc pas possible en général de visualiser l'entièreté de la MEMOIRE CENTRALE. Aussi **PROGRAIS** sélectionne une section de la MEMOIRE CENTRALE qui est rendue visible dans la fenêtre qui apparaît à l'écran. Cette sélection est faite de la façon illustrée dans la figure de la page suivante (*Fig. 3*):

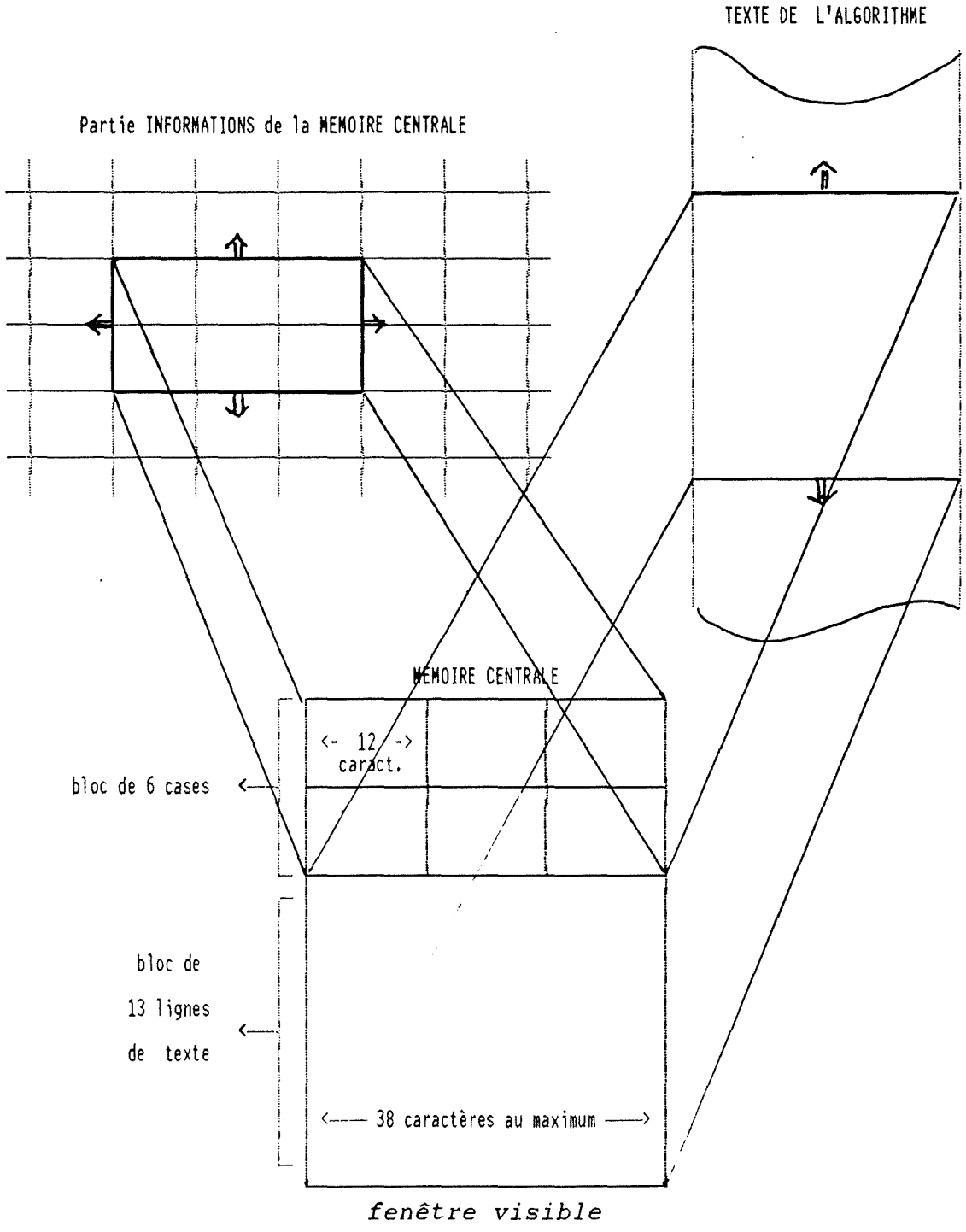


Fig. 3 Principes de la sélection de la section de la Mémoire Centrale visible à l'écran (les flèches indiquent les déplacements possibles)

Pour la représentation du contenu d'une case de la partie de la MEMOIRE CENTRALE réservée aux informations et aux résultats, **PROGRAIS** procède de la façon suivante (*Fig. 4*):

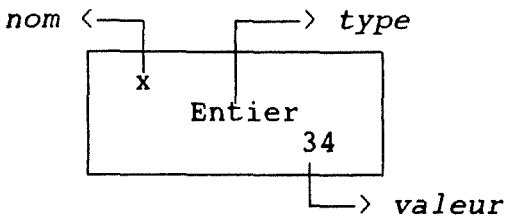


Fig. 4 case de nom "a", de type "entier" et de valeur "34"

Notons que la disposition du nom et de la valeur diffère légèrement de ce qui est défini dans le **Modèle Général d'un Ordinateur**. De plus, **PROGRAIS** indique également le type de la case.

A la place du signet "->", **PROGRAIS** schématise les circuits gérant l'ordre d'exécution en plaçant en "Vidéo inversée" la ligne de texte dont le contenu est en cours d'exécution.

Contexte d'utilisation de PROGRAIS

PROGRAIS permet d'exécuter pas-à-pas des programmes (ou algorithmes) simples conçus par l'utilisateur ou choisis dans une bibliothèque d'exemples. Pour être exécutés, les programmes sont en principe des programmes corrects. En cas d'erreurs de construction, **PROGRAIS** indique la cause d'erreur mais il ne prend pas en charge la correction.

Dans la version actuelle de **PROGRAIS**, le langage de programmation utilisé pour représenter les programmes est une version simplifiée du *langage Pascal* standard appelée *DEMO PASCAL* contenant tous les concepts fondamentaux du langage

Pascal standard. Les informations prises en compte seront du type **entier** ou du type **booléen** et les variables "tableau" sont autorisées. Pour le reste, les concepts retenus sont essentiellement les concepts décrits dans la *Méthode d'Initiation aux Raisonnements de la Programmation* de R. Lesuisse et A. Borsu[4].

PROGRAIS travaille comme un Interpréteur de programmes au sens informatique, ce qui veut dire qu'il effectue, les unes après les autres, les différentes instructions et opérations qui composent le programme (ou algorithme) tout comme cela se fait en général en langage BASIC sans passer par une représentation intermédiaire compilée.

Matériel requis

PROGRAIS peut être employé sur n'importe quel ordinateur "PC compatible" admettant au moins le mode monochrome ou le mode CGA. Il fonctionne aussi bien sur écran noir et blanc que sur écran couleur.

Il est à remarquer que, comme **PROGRAIS** est destiné à des étudiants, il a été conçu pour pouvoir être utilisé avant tout sur un matériel de base qui est le matériel en général mis à la disposition des étudiants (c.-à-d. des "PC Compatible" avec processeurs de type 8086 ou 8088). Il fonctionne évidemment tout aussi bien sur des "PC Compatible" plus performants.

Enfin, pour améliorer la qualité de l'image et la rapidité d'affichage, **PROGRAIS** travaille simplement en mode texte.

---

(4) On trouvera plus de détails concernent le langage reconnu dans l'ouvrage de R. Lesuisse et A. Borsu déjà cité ainsi que dans le Chapitre 3 de ce mémoire.

Ce qu'on trouve sur la disquette

La disquette fournie avec **PROGRAIS** contient le fichier

PROGRAIS.COM

qui permet de faire fonctionner le logiciel. La disquette contient de plus différents fichiers de programmes de démonstration écrit en Pascal. Ces fichiers sont les suivants:

- les fichiers ESS0.PAS à ESS8.PAS représentant des programmes corrects,

ainsi que

- les fichiers ESSVIDE.PAS, ESSPROG.PAS, ESSDUPLI.PAS et ESSLONG.PAS représentant des programmes contenant des erreurs de construction.

Comment démarrer PROGRAIS ?

Si vous voulez démarrer PROGRAIS à partir de la disquette fournie, tapez:

PROGRAIS

au message A> ou B> suivant le lecteur de disquette utilisé.

Si vous voulez démarrer PROGRAIS à partir d'un disque dur, il vous suffit alors de copier tous les fichiers de la disquette dans le répertoire du disque dur de votre choix en tapant

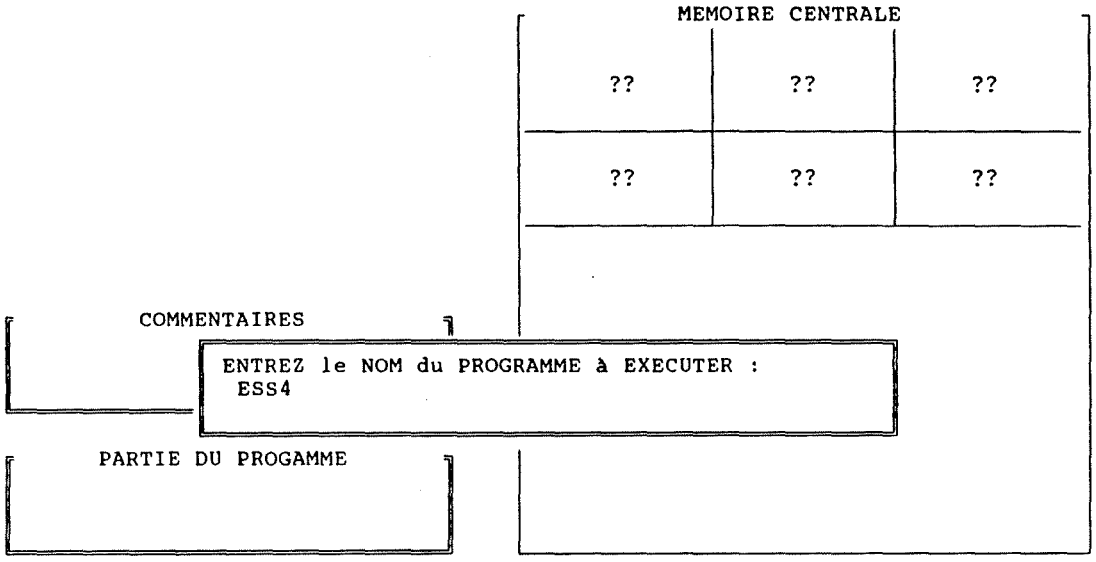
COPY *.* C\MONREP.

Il faut évidemment que le répertoire MONREP existe. Sinon, il faut créer un répertoire avant de copier les fichiers. Pour démarrer PROGRAIS, il vous suffit alors de vous placer dans le répertoire choisi et de taper

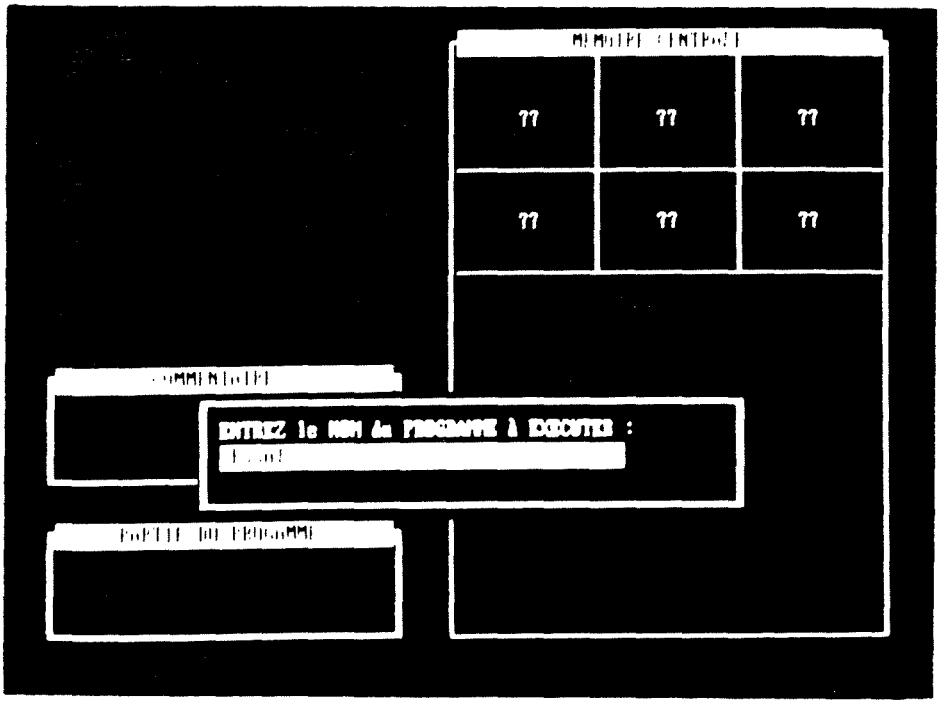
PROGRAIS.

Mode d'emploi de PROGRAIS

Au départ, l'écran de PROGRAIS montre, en arrière plan, la représentation du **Modèle Général d'un Ordinateur** et, en avant plan, une fenêtre permettant de choisir le programme à exécuter. Cela donne, en copie d'écran



<ESC> permet de parcourir les cases "INFORMATIONS"

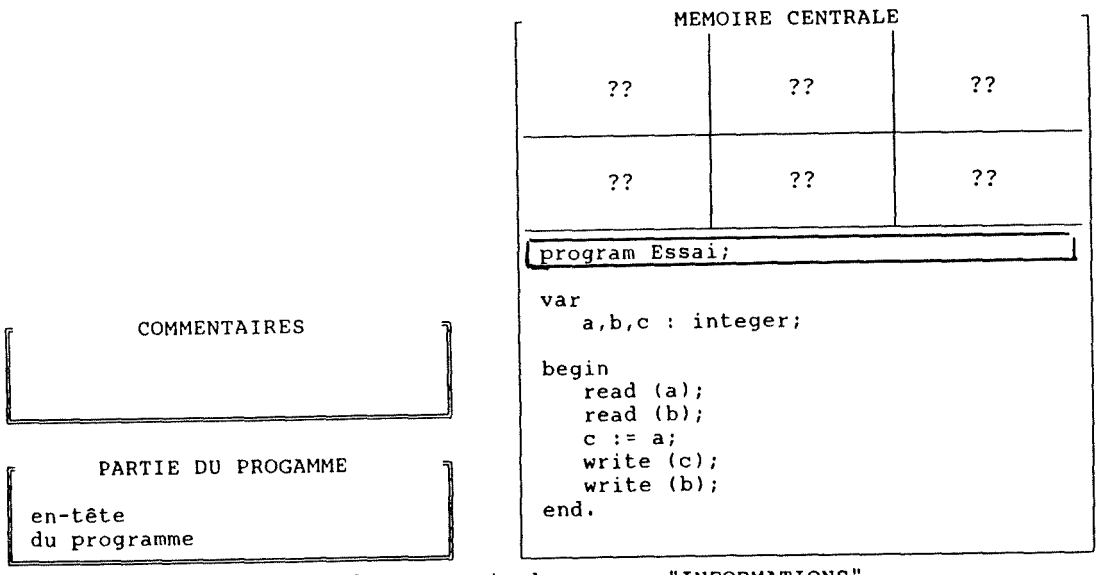


Le nom du fichier est donc introduit à la demande. L'extension ".PAS" est l'extension par défaut. Pour les fichiers qui possèdent cette extension, il n'est pas nécessaire de taper ".PAS". Dans le cas où le nom de fichier introduit n'existe pas, il y a renouvellement de la demande.

Une fois le nom introduit, **PROGRAIS** place le texte du programme dans le partie des opérations de la MEMOIRE CENTRALE et les 13 premières lignes du texte sont montrée à l'écran. Commence alors l'exécution proprement dite. L'avancement dans l'exécution se fait à la demande en tapant sur une touche quelconque. Ceci permet d'observer à sa guise les modifications qui s'effectuent dans la MEMOIRE CENTRALE ou sur l'ECRAN. Lorsqu'il y a des modifications dans l'ECRAN, la fenêtre visualisant la MEMOIRE CENTRALE disparaît pour faire apparaître l'ECRAN en entier.

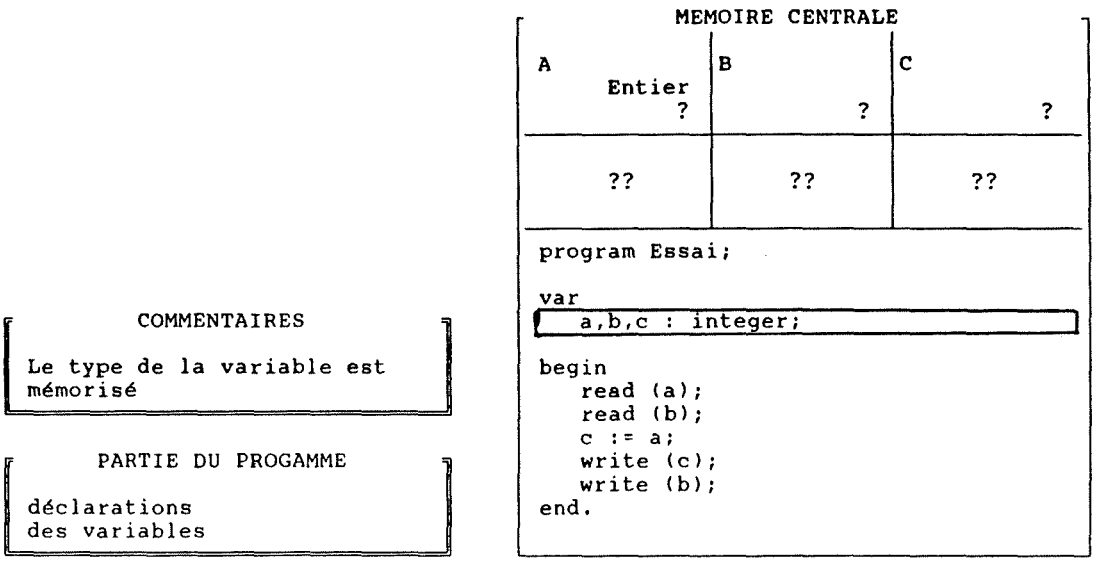
Les quelques écrans qui suivent permettent d'observer différents stades de l'exécution d'un problème dans **PROGRAIS**. (L'exemple considéré est contenu dans le fichier ESS4.PAS).

Ecran n° 1 :



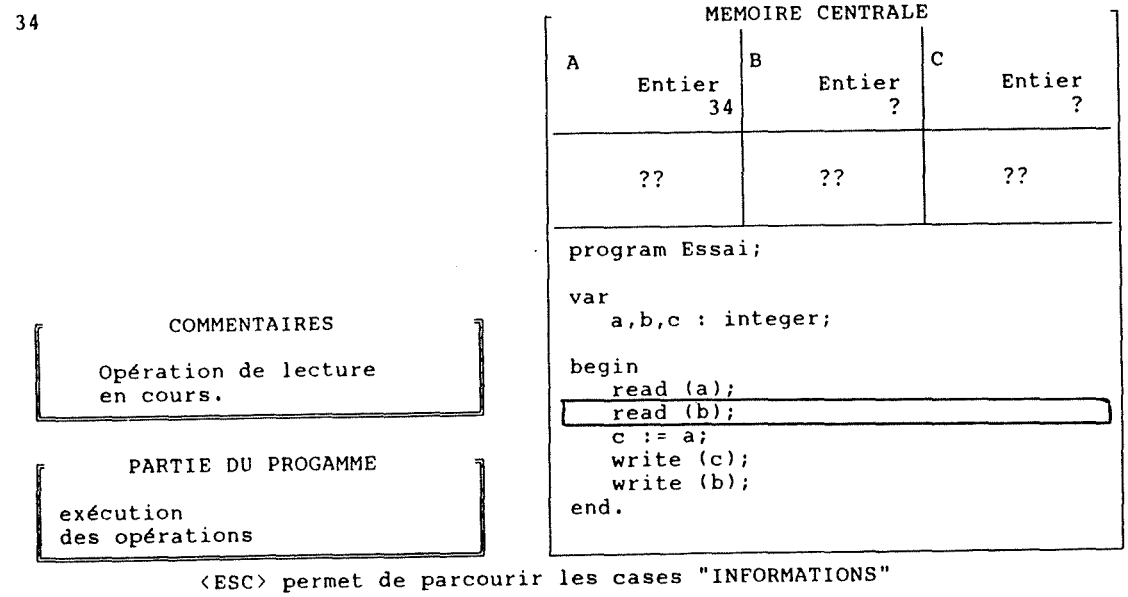
<ESC> permet de parcourir les cases "INFORMATIONS"

Ecran n° 2 :



<ESC> permet de parcourir les cases "INFORMATIONS"

Ecran n° 3 :



<ESC> permet de parcourir les cases "INFORMATIONS"

Ecran n° 4 :

3492

<ESC> permet de parcourir les cases "INFORMATIONS"

Ecran n° 5 :

3492

MEMOIRE CENTRALE

A	B	C
Entier 34	Entier 92	Entier 34
??	??	??

```
program Essai;

var
   a,b,c : integer;

begin
   read (a);
   read (b);
   c := a;
   write (c);
   write (b);
end.
```

COMMENTAIRES

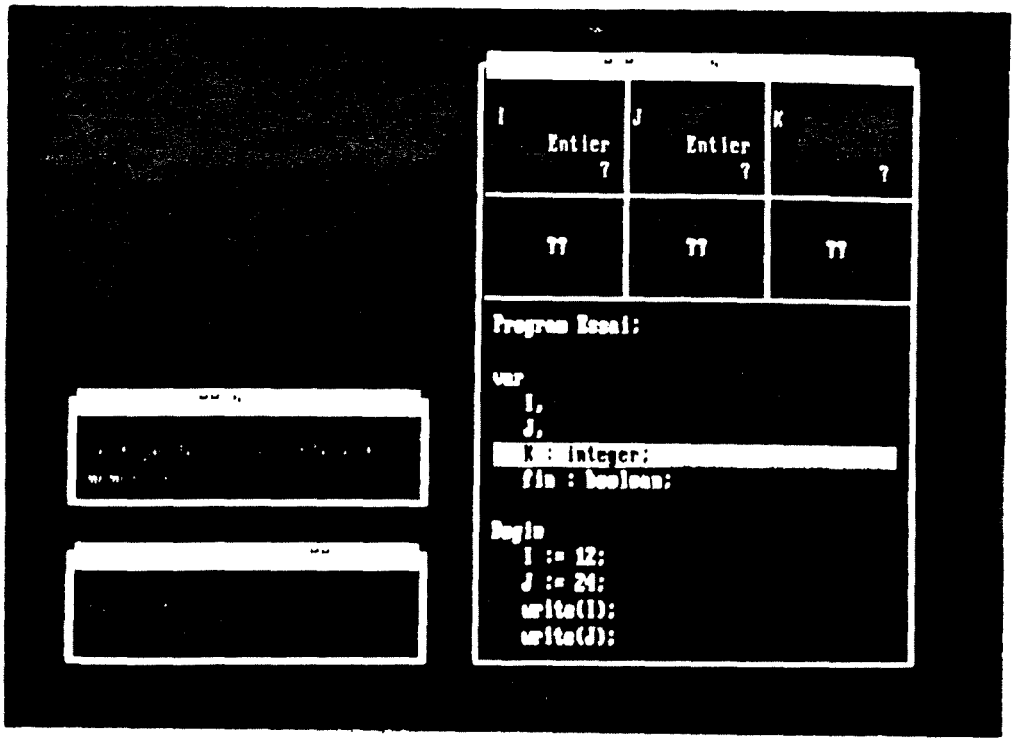
Opération d'affectation en cours.

PARTIE DU PROGAMME

exécution des opérations

<ESC> permet de parcourir les cases "INFORMATIONS"

et, en photographie



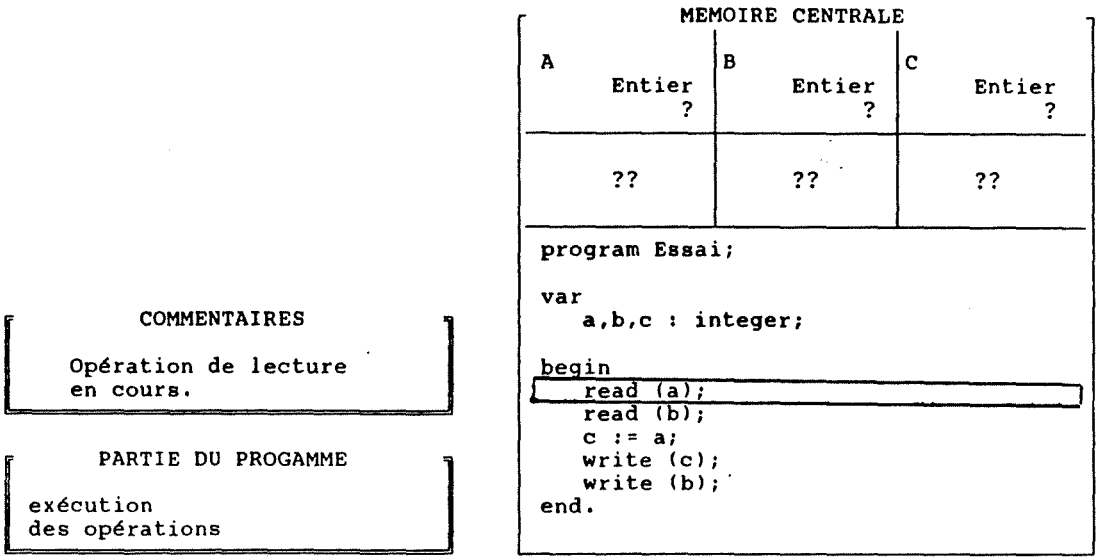
Il est très important de bien réaliser la distinction qui existe entre l'ECRAN faisant partie du **Modèle Général d'un Ordinateur** et l'écran de l'ordinateur que **PROGRAIS** utilise pour faire l'exécution.

Les deux fenêtres supplémentaires de titre COMMENTAIRES et PARTIE DE PROGRAMME donnent des informations relatives à l'exécution du programme. La fenêtre PARTIE DU PROGRAMME situe la partie du programme et la fenêtre COMMENTAIRES donne des informations sur la nature de l'instruction en train d'être exécutée.

Dans le bas de l'écran de PROGRAIS se trouve la ligne:

<ESC> permet de parcourir les cases "INFORMATIONS"

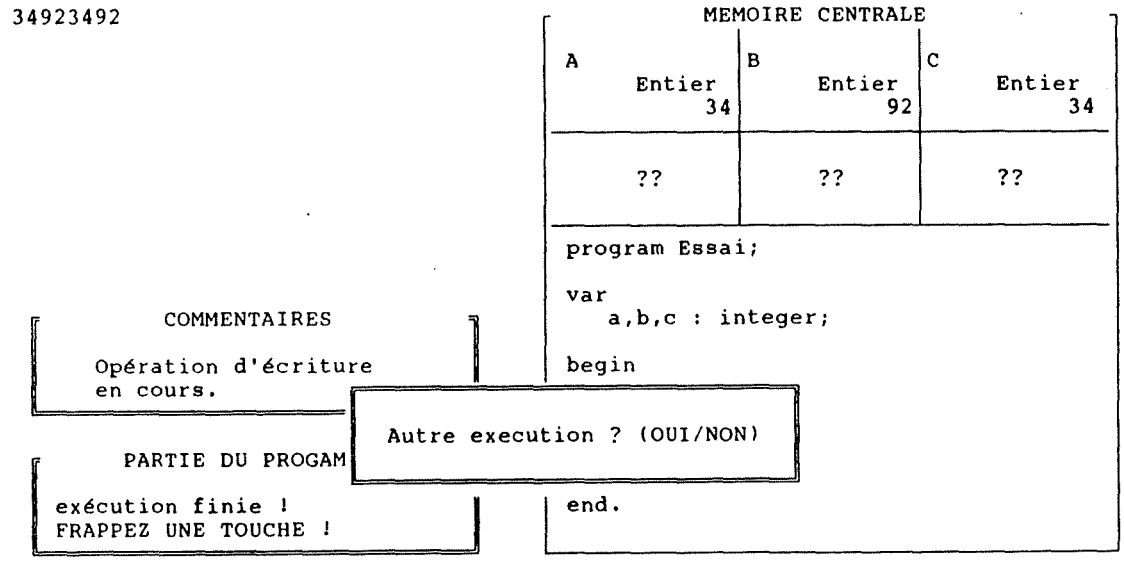
Cette option permet d'interrompre l'exécution à tout moment. On obtient alors l'écran suivant:



↑et↓ : défilement haut et bas - <ESC> : Retour Exécution

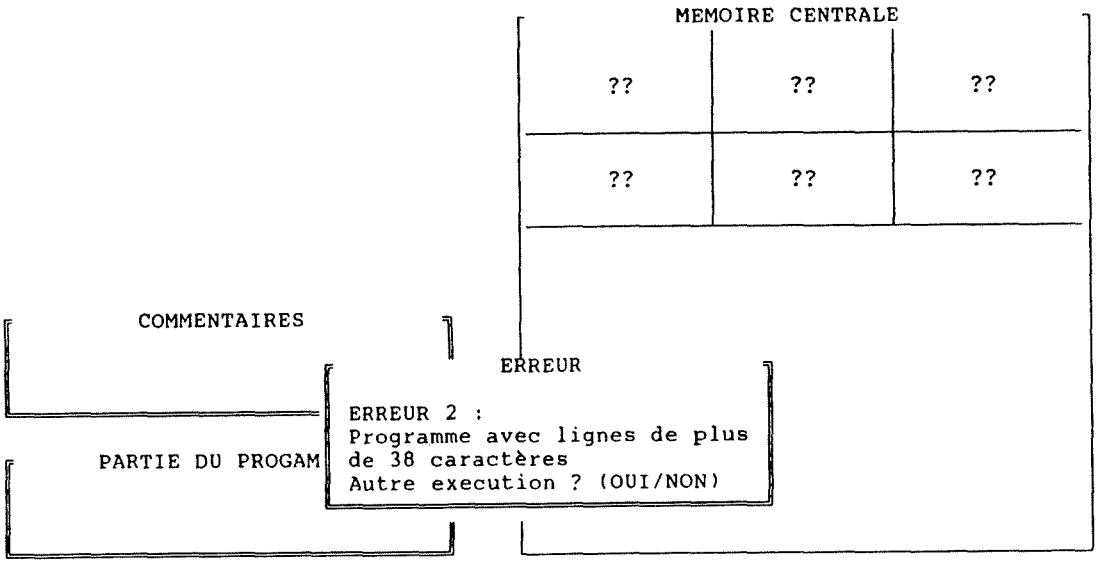
Il est alors possible de déplacer la fenêtre visible dans l'ensemble de la MEMOIRE CENTRALE. Les déplacements se font alors en utilisant les flèches "haut" et "bas" et il y a un signal sonore si les déplacements ne sont plus possibles. La touche <ESC> permet de reprendre le cours de l'exécution.

En fin d'exécution, il est alors proposé de recommencer une nouvelle exécution ou non:

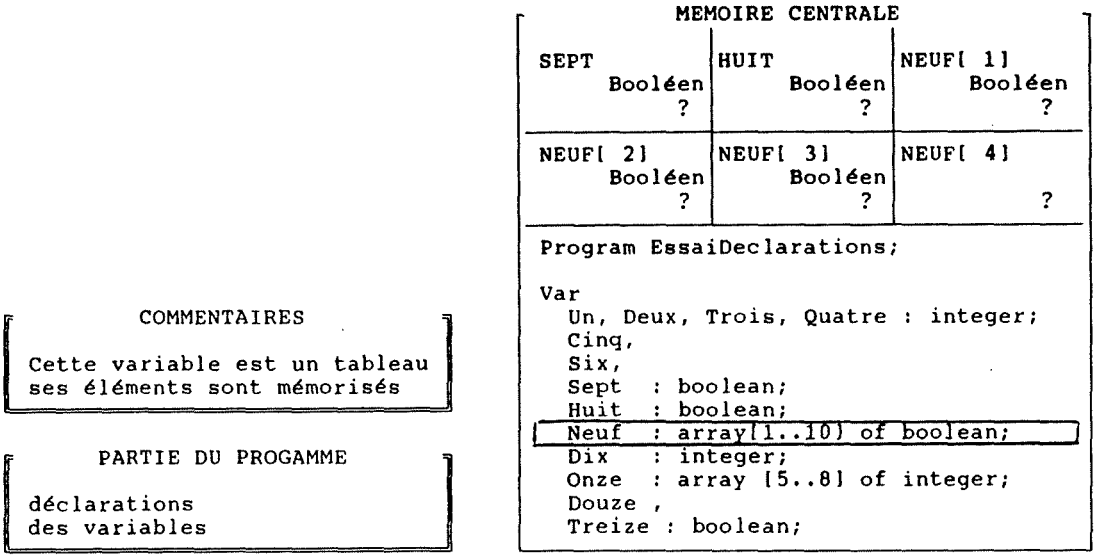


<ESC> permet de parcourir les cases "INFORMATIONS"

Si le programme introduit ne peut pas être exécuté, alors PROGRAIS indique le type d'erreur et propose de recommencer une autre exécution ou non. Les erreurs peuvent être de deux types, soit que le programme contient une erreur relative au langage Pascal en général, soit que le programme contient des instructions de langage Pascal qui ne sont pas prises en charge par PROGRAIS. Il ne faut pas oublier que PROGRAIS ne permet d'exécuter que des programmes élémentaires. Dans le cas d'un programme avec erreur, l'écran se présente ainsi (programme ESSLONG.PAS):



PROGRAIS peut également exécuter de programmes contenant des variables "tableau". Dans ce cas, les différents éléments du tableau sont introduits les uns après les autres dans la partie "informations" de la MEMOIRE CENTRALE comme on peut le voir dans l'exemple d'écran qui suit (programme ESS7.PAS):



<ESC> permet de parcourir les cases "INFORMATIONS"

4.3 Conclusion.

Grâce à ce petit guide de l'utilisateur, l'outil est apte à être utilisé et le chemin est bouclé. Le projet est une réalité. Cependant, la version proposée n'est en fait qu'un prototype et elle n'a pas été testée par des utilisateurs potentiels. C'est le début de la phase de la maintenance. Mais c'est une autre histoire!

CONCLUSION

Nous voilà arrivée au terme de cette exposition à propos du développement d'un outil informatique annexe à la *Méthode d'Initiation aux Raisonnements de la Programmation* de Lesuisse et Borsu [A/LE87]. Dans les quatre chapitres qui précèdent, nous n'avons pu que survoler tous les concepts mis en jeu dans un tel développement.

En guise de conclusion, nous nous proposons de faire quelques commentaires tirés du travail, sur la version actuelle de l'outil **PROGRAIS** mais aussi sur la place de l'outil dans le domaine de l'enseignement de la Programmation.

L'outil et la version actuelle.

La version actuelle de l'outil **PROGRAIS** est encore bien imparfaite mais, grâce à la démarche de développement qui a été retenue, la rendre plus performante ne doit pas poser de problèmes majeurs. Les améliorations à apporter se situent à deux niveaux.

Tout d'abord, les objectifs de départ n'ont pas encore été entièrement satisfaits. En effet, la version actuelle de l'outil ne permet pas de prendre en charge tous les concepts définis dans le langage *DEMO PASCAL*. Il reste donc encore à parachever le travail entrepris surtout au niveau du module *EXECUTEUR DE PROGRAMMES* de l'architecture logique. Mais les grandes options permettant d'intégrer l'exécution dans le cadre du **Modèle Général d'un Ordinateur** de Lesuisse et Borsu

existent. Donc, à ce niveau, les améliorations sont en fait une question de temps.

Les améliorations à apporter se situent également à un autre niveau. Dans la version proposée, certains choix de présentation ont été faits, par exemple au niveau des fenêtres d'aides. De même, dans cette version, il existe une ébauche de choix d'options dans la ligne de bas d'écran. Ces différents choix peuvent certainement être améliorés. Cela demande de faire une expertise non seulement auprès des concepteurs de la méthode mais aussi auprès d'étudiants utilisateurs.

L'outil et l'enseignement de la Programmation.

En filigrane de ce travail, nous nous sommes posés la question de savoir ce que l'outil pouvait apporter de plus à la *Méthode d'Initiation aux Raisonnements de la Programmation* de Lesuisse et Borsu [A/LE87]. Nous sommes convaincues que la méthode de Lesuisse et Borsu présente surtout le grand intérêt d'obliger les étudiants à raisonner par abstraction, ce qui est, à notre sens indispensable dans toute démarche intellectuelle. Aussi, nous pensons que l'outil peut trouver sa place essentiellement auprès des étudiants qui ne possèdent pas ce sens de l'abstraction en tout début de la phase d'apprentissage.

Dans un même ordre d'idées, nous nous sommes interrogée sur la place que pouvait prendre l'outil dans le contexte de l'enseignement assisté par ordinateur (EAO). Dans la littérature relative à l'apprentissage de la programmation citée dans la section B de la BIBLIOGRAPHIE, nous avons trouvé plusieurs tentatives utilisant l'ordinateur comme support. Cependant, dans la plupart des cas les systèmes développés sont des systèmes du type *questions/réponses* basé sur le dialogue entre l'ordinateur et l'élève. L'outil PROGRAIS comporte des caractéristiques différentes. L'apprentissage se fait par observation en regardant comment l'ordinateur procède pour exécuter l'algorithme. L'étudiant peut ainsi comprendre par lui-même le pourquoi de ces erreurs.

Enfin, nous avons réfléchi sur ce qu'un outil comme **PROGRAIS** peut apporter de plus qu'un bon "débugeur" proposé actuellement dans tous les compilateurs. En fait, les objectifs du débugeur sont totalement différents. Le débugeur est un outil d'aide à la construction d'algorithmes et il n'est pas conçu pour apprendre. Il ne peut donc fournir cette dimension d'enseignement assisté par ordinateur présente dans l'outil **PROGRAIS.**

L'outil PROGRAIS trouve donc bien sa place dans un contexte d'enseignement assisté par ordinateur pour l'enseignement de la Programmation et mérite qu'on y prête attention.

BIBLIOGRAPHIE

A. Ouvrage de Référence.

A/LE87 Lesuisse R. et Borsu A., *Initiation aux Raisonnements de la Programmation*, Presses Universitaires de Namur, 1987(1).

B. Apprentissage de la Programmation.

B/BO84 Boehm B.W., Gray T.E. et Seewaldt T, *Prototyping Versus Specifying: A Multiproject Experiment*, IEEE Transactions on Software Engineering SE.10 N°3, 1984, pp. 290-302, **reproduit dans** *Tutorial: Human Factors in Software Development*, édité **par** B. Curtis, IEEE Computer Society, 1986, pp. 298-311.

B/CL87 Clancey W.J., *Intelligent Tutoring Systems: A Tutorial Survey*, **dans** *Current Issues In Expert Systems*, édité par A. van Lamsweerde et P. Dufour, Academic Press, 1987, pp. 39-78.

B/CU80 Curtis B., *Measurement and Experimentation in Software Engineering*, Proceedings of the IEEE 68 N°9, 1980, pp. 1144-1157, **reproduit dans** *Tutorial: Human Factors in Software Development*, édité **par** B. Curtis, IEEE Computer Society, 1986, pp. 717-730.

(1) actuellement en cours de réédition.

B/DA87 Darimont R. et Milgrom E., *Teaching Programming by Induction*, IX International Symposium "Computer at the University", Cavtat, 1987(2).

B/DI86 diSessa A.A. et Abelson H., *Boxer: a Reconstructible Computational Medium*, Communications of the ACM 29 N°9, 1986, pp. 859-868.

B/DU83 Duchateau C., *Programmer: Pour une Découverte des méthodes de la Programmation*, Wesmael-Charlier, 1983.

B/GO82 Goldstein I.P., *The Genetic Graph: a Representation for the Evolution of Procedural Knowledge*, **dans** *Intelligent Tutoring Systems*, édité par D. Sleeman et J.S. Brown, Academic Press, 1982, pp. 51-77.

B/HO87 Holland J.H., Holyoak K.J., Nisbett R.E. et Thagard P.R., *Induction: Process of Inference, Learning and Discovery*, MIT Press, 2ème édition, 1987, chapitre 1.

B/JO86 Johnson W.L., *Intention-Based Diagnosis of Novice Programming Errors*, Pitman, 1986.

B/LE73 Leclercq D., Donnay J., De Bal R. et Lambrecht P., *Construire un Cours Programmé*, Laboratoire de Pédagogie expérimentale de l'Université de Liège, Nathan et Labor, 1973.

B/MA79 Mayer R., *A Psychology of Learning BASIC*, Communications of the ACM 22 N°11, 1979, pp. 589-593.

B/MA81a Mayer R., *The Psychology of How Novices Learn Computer Programming*, Computing Surveys 13 N°1, 1981, pp. 121-141.

---

(2) Nous remercions Madame Anne Borsu de nous avoir communiqué cette référence.

B/MA81b Mayer R.E. et Bayman P., *Psychology of Calculator Languages: A Framework for Describing Differences in Users' Knowledge,* Communications of the ACM 24 N°8, 1981, pp. 511-520, **reproduit dans** *Tutorial: Human Factors in Software Development,* édité **par** B. Curtis, IEEE Computer Society, 1986, pp. 432-441.

B/MA86 Mayer R.E., Dyck J.L. et Vilberg W., *Learning to Program and Learning to Think: What's the Connection?,* Communications of the ACM 29 N°7, 1986, pp. 605-610.

B/MI82 Miller M.L., *A Structured Planning and Debugging Environment for Elementary Programming,* **dans** *Intelligent Tutoring Systems,* édité par D. Sleeman et J.S. Brown, Academic Press, 1982, pp. 51-77.

B/NA90 Nachmias R., Friedler Y. et Linn M.C., *The Role of Programming Environments in Pascal Instruction,* Computers Educ. 14 N°2, 1990, pp. 145-158.

B/SH81 Shneiderman B. et Mayer R., *Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results,* International Journal of Computers and Information Sciences 8 N°3, 1979, pp. 219-238, **reproduit dans** *Tutorial: Human Factors in Software Development,* édité **par** B. Curtis, IEEE Computer Society, 1986, pp. 10-24.

B/SL86 Sleeman D., *The Challenges of Teaching Computers Programming,* Communications of the ACM 29 N°9, pp. 840-841.

B/SH81 Sheil B.A., *The Psychological Study of Programming,* Communications of the ACM 13 N°1, 1981, pp. 101-120, **reproduit dans** *Tutorial: Human Factors in Software Development,* édité **par** B. Curtis, IEEE Computer Society, 1986, pp. 689-708.

B/SO86 Soloway E., *Learning to Program = Learning to Construct Mechanisms and Explanations*, Communications of the ACM 29 N°9, 1986, pp. 850-858.

B/SP85 Spohrer J.C., Soloway E. et Pope E., *A Goal/Plan Analysis of Bugger Pascal Programs*, Human-Computers Interaction 1 N°2, 1985, pp. 163-207, **reproduit dans** *Tutorial: Human Factors in Software Development*, édité **par** B. Curtis, IEEE Computer Society, 1986, pp. 535-579.

B/SP86 Spohrer J.C. et Soloway E., *Novice Mistakes: Are the Folk Wisdoms Correct?*, Communications of the ACM 29 N°9, 1986, pp. 624-632.

C. Interactivité de l'Edition.

C/LU82 *La Réalisation des Logiciels Graphiques Interactif (Ecole d'été d'Informatique, CEA-EDF-INRIA, juillet 1979)* publié **sous la direction** de M. Lucas, Eyrolles, 1982.

C/ME82 Meyrowitz N. et Van Dam A., *Interactive Editing Systems: Part I*, Computing Surveys 14 N°3, 1982, pp. 321 et suivantes.

C/PE80 Peterson J.L., *Computer programs for spelling correction: an experiment in program design*, Springer-Verlag, 1980.

D. Méthodologie de Développement de Logiciels.

D/DI68 Dijkstra E.W., *The Structure of "THE"-Multiprogramming System*, Communications of the ACM 11 N°5, 1968, pp. 341-346.

D/ME88 Meyer B., *Object-Oriented Sofware Construction*, Prentice-Hall, 1988, Chapitre 1.

D/VA87 van Lamsweerde A., Cours de *Méthodologie de développements de logiciels* de 2ème Licence et Maîtrise en Informatique des Facultés Universitaires de la Paix, Namur, 1987-88.

E. Algorithmique et Langages.

E/GO86 Goldschlager L. et Lister A., *Informatique et Algorithmique*, InterEditions, 1986 (édition originale en anglais sous le titre *Computer Science: A Modern Introduction*, Prentice-Hall, 1982).

E/GR81 Gries D., *The Science of Programming*, Springer-Verlag, 1981.

E/MA87 Marcotty M. et Ledgard H., *The World of Programming Languages*, Springer-Verlag, 1987.

E/SA69 Sammet J.E., *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969.

E/TL87 *Les langages de programmation* **dans** la collection *Le Monde des ordinateurs* **par** les Rédacteurs des Editions Time-Life, 1987.

E/WI77 Wirth N., *Introduction à la Programmation Systématique*, Masson, 1977 (2ème tirage: 1981).

F. Théorie des Programmes.

F/AH86 Aho A., Sethi R. et Ullman J., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986. (Version française: *Compilateurs: Principes, Techniques et Outils*, InterEditions, 1989.

F/BE85 Beck L.L., *SYSTEM SOFTWARE: An Introduction to Systems Programming*, Addison-Wesley, 1985, Chapitre 5, pp. 207-294.

F/HE80 Henderson P., *Funtional Programming, Application and Implementation*, Prentice-Hall, 1980, Chapitres 11-12.

F/KN68 Knuth D.E., *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 N°2, 1968, pp. 127-145.

F/LE88 Le Charlier B., Cours de *Théorie des programmes* de 2ème Licence et Maîtrise en Informatique des Facultés Universitaires de la Paix, Namur, 1987-88.

F/LI78 LIVERCY C.(nom collectif), *Théorie des programmes, Schémas, Preuves, Sémantique*, Dunod, 1978, Chapitre 5.

F/LU84 Lucas M., *Algorithmique et Représentation de Données: 2. Evaluations, Arbres, Graphes, Analyse de Textes*, Masson, Paris, 1984.

F/TE81 Tennent R.D., *Principles of Programming Languages*, Prentice-Hall, 1981, Chapitres 1, 2 et 13.

F/WE80 Welsh J. et McKeag M., *Structured System Programming*, Prentice-Hall, 1980, Section 2.

F/WI76 Wirth N., *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976, Chapitre 5.

F/WI81 Wirth N., *Pascal-S: A Subset and its Implementation*, **dans** *Pascal - The Language and its Implementation*, édité par D.W. Barron, John Wiley & Sons, 1981, pp. 199-259.

G. Langage Pascal.

G/BOxx Borland, *Manuels de Référence de TURBO Pascal, Versions 3.0, 4.0 et 5.5.*

G/CA86 Cardinael J-P, Cours de *Pascal* de 1ère Licence et Maîtrise en Informatique des Facultés Universitaires de la Paix, Namur, 1986-87.

G/CO85 Cooper D. et Clancy M., *Oh! Pascal! (second edition)*, Norton & Company, 1985.

G/JE78 Jensen K. et Wirth N., *Pascal, User manual and report*, Springer-Verlag, 1978.

G/LE85 Lecarme O. et Nebut J-L., *PASCAL pour programmeurs*, McGraw-Hill (Paris), 1985.

G/LE79 Ledgard H.F., Nagin P. et Hueras J., *Pascal with Style: Programming Proverbs*, Hayden Book Co, 1979.

H. Techniques Avancées en Turbo Pascal.

H/ME87 Meyer J-J., *Pratique du Turbo Pascal, Créez vos Progiciels*, Editions Radio, 1987.

H/OB88 O'Brien S.K., *Turbo Pascal: The Complete Reference*, Borland-Osborne/McGraw-Hill (Programming Series), 1988.

I. Ouvrages Généraux.

I/RO67 *Le Petit Robert*, Paris 1967 (édition 1982).

ANNEXES

Annexe 1

PETITS PROGRAMMES DE DEMONSTRATION

A.1 Liste des programmes de démonstration.

ESS0.PAS

programme le plus simple contenant seulement les instructions **Begin** et **End.**

ESS1.PAS

programme contenant la déclaration d'une variable entière et une opération d'affectation de valeur pour cette variable.

ESS2.PAS

programme contenant les déclarations de plusieurs variables entières et booléennes suivies de plusieurs opérations d'affectation et d'écriture (avec écriture d'une variable déclarée mais à laquelle aucune valeur n'a été affectée).

ESS3.PAS

programme avec la déclaration d'une variable avec un nom de plus de 12 caractères.

ESS4.PAS

programme contenant des opérations d'affectations, d'écriture et de lecture portant sur des variables entières et booléennes.

ESS5.PAS
programme semblable au précédent mais avec plus de lignes de texte.

ESS6.PAS
programme avec une déclaration d'une variable tableau.

ESS7.PAS
programme avec des déclarations multiples de variables entières et booléenne et des variables tableaux.

ESS8.PAS
programme semblable aux programmes ESS4 et ESS5 avec plus de lignes de texte.

ESS9.PAS
programme *Sommation* du livre de Lesuisse et Borsu [A/LE87] pris en exemple dans le premier chapitre.

Les autres petits programmes proposés sont des programmes contenant des erreurs soit par rapport au langage Pascal standard soit par rapport au mini langage *DEMO PASCAL* retenu pour l'outil:

ESSVIDE.PAS
programme vide.

ESSPROG.PAS
programme sans le mot réservé **Program** en début (erreur dans le langage Pascal standard).

ESSDUPLI.PAS
programme avec la déclaration de deux variables de même nom (erreur dans le langage Pascal standard).

ESSLONG.PAS
programme contenant des lignes de codes avec plus de 38 caractères (erreur dans le *DEMO PASCAL*).

A.2 Codage des programmes de démonstration.

ESS0.PAS :

```
Program Essai;

Begin

End.
```

ESS1.PAS :

```
Program Essai;

var
   Essai : integer;

Begin
   Essai := 1244
End.
```

ESS2.PAS :

```
Program Essai;

var
   I,
   J,
   K : integer;
   fin : boolean;

Begin
   I := 12;
   J := 24;
   write(I);
   write(J);
   write(K);
End.
```

ESS3.PAS :

```
Program Essai;
```

```
var
   NomTresTresTresLong : boolean;

begin

end.
```

ESS4.PAS :

```
program Essai;

var
   a,b,c : integer;

begin
   read (a);
   read (b);
   c := a;
   write (c);
   write (b);
end.
```

ESS5.PAS :

```
Program Essai;

var
   I,
   J,
   K : integer;
   fin : boolean;

Begin
   I := 12;
   J := 24;
   write(I);
   write(J);
   write(I,J);
   I := J;
   write(I,J);
   write(J,I);
   write(I,J);
   fin := true;
   write(fin);
   fin := false;
   write(fin);
End.
```

ESS6.PAS :

```
program decTab;

var
   tableau : array [1..12] of integer;

begin

end.
```

ESS7.PAS :

```
Program EssaiDeclarations;

Var
  Un, Deux, Trois, Quatre : integer;
  Cinq,
  Six,
  Sept   : boolean;
  Huit   : boolean;
  Neuf   : array[1..10] of boolean;
  Dix    : integer;
  Onze   : array [5..8] of integer;
  Douze ,
  Treize : boolean;

Begin

End.
```

ESS8.PAS

```
Program Essai;

var
   I,
   J,
   K : integer;
   fin : boolean;

Begin
   I := 12;
   J := 24;
   write(I);
   write(J);
   I := J;
   write(I);
   write(J);
   write(K);
```

```
   K := I;
   write(I);
   write(J);
   write(K);
   K := 49;
   write(I);
   write(J);
   write(K);
   read(I);
   write(I);
End.
```

ESS9.PAS :

```
program sommation;

var
   a,b,c : integer;

begin
    read (a);
    read (b);
    c := a + b;
    write (c);
end.
```

ESSVIDE.PAS :

ESSPRG.PAS :

```
Var
     Essai : integer;

Begin
     Essai := 12;
End.
```

ESSDUPLI.PAS

```
Program Essai;
```

```
var
   Essai : integer;
   Essai : integer;
   Essai : boolean;

Begin
   Essai := 1244
End.
```

ESSLONG.PAS :

```
program esslong;

begin
   write(' Ce programme contient une ligne code
trop longue ');
end.
```

Annexe 2

CODAGE

A.1 Codage des sous-sytèmes successifs.

A.1.1 Premier sous-système : PRAIS01.PAS.

```
{Listage de PRAIS01.PAS}
Program Prograis01;
{*************************************************************************}
{*                                                                       *}
{*                  PROGRAMME de Demonstration no.1                      *}
{*                                                                       *}
{*                        PREMIER SOUS-SYSTEME                           *}
{*                        de l'outil PROGRAIS                            *}
{*                                                                       *}
{*              *** ETABLISSEMENT d'un ENVIRONNEMENT ***                 *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{*              AUTEUR : Anne de Baenst-Vandenbroucke                    *}
{*                                                                       *}
{*************************************************************************}

{
-------------------------------------------------------------------------
```

PREMIER SOUS-SYSTEME :

Ce premier sous-systeme etablit le decoupage de l'ecran reel en quatre fenetres :

- une pour l'ECRAN,
- une pour la MEMOIRE CENTRALE,
- une pour des MESSAGES eventuels,
- une, horizontale, a la ligne du bas, pour un MENU.

Il illustre egalement la gestion de ces differentes fenetres.

Ce premier sous-systeme "UTILISE" les modules suivants:
- Librairie de base,
- Librairie de fenetrage,
- gerant des messages.

La primitive "InitDecoupage" est une ebauche pour le module :
- controleur de communication.

```
-------------------------------------------------------------------------}

{ fichiers inclus : }
{$I Defgloba.lib}  { declarations globales pour les librairies}
{$I Routines.lib}  { librairie de base }
{$I Winprimi.lib}  { librairie de fenetrage }

{----------------------}
Procedure InitDecoupage;
{----------------------}
{<<<
  Etablit un environnement de fenetrage suivant les principes de
  PROGRAIS.
>>>}
var
   I       : integer;
   Chaine  : str78;
   Ch      : char;

Begin
   TextMode;
   TextBackGround (black);
   TextColor (yellow);
   Curseur (FALSE);

   with Fenetres [1] do    { fenetre pour simuler l'ECRAN }
     begin
       XHG := 0;
       YHG := 0;
       XBD := 79;
       YBD := 23;
       SavBuf := Nil;
       Tracee := FALSE;
       Cadre := FALSE;
       Move (SimplTrait, Bord, 6);
       Titre := '';
       SavCoul := yellow;
       CoulCad := CoulCadre;
       SavX := 0;
       SavY := 0;
     end;  { fenetre 1 }

  with Fenetres [2] do     { fenetre pour simuler la MEMOIRE CENTRALE }
     begin
       XHG := 38;
       YHG := 0;
       XBD := 79;
       YBD := 23;
       SavBuf := Nil;
       Tracee := FALSE;
       Cadre := TRUE;
       Move (SimplTrait, Bord, 6);
       Titre := 'MEMOIRE CENTRALE';
       SavCoul := lightred;
       CoulCad := CoulCadre;
       SavX := 0;
       SavY := 0;
```

```
    end;  { fenetre 2 }

  with Fenetres [4] do    { fenetre MENU ( ligne 25 ) }
    begin
      XHG := 0;
      YHG := 24;
      XBD := 79;
      YBD := 25;
      SavBuf := Nil;
      Tracee := FALSE;
      Cadre := FALSE;
      Move (SimplTrait, Bord, 6);
      Titre := '';
      SavCoul := white;
      CoulCad := CoulCadre;
      SavX := 0;
      SavY := 0;
    end;  { fenetre 4 }

  with Fenetres [5] do    { fenetre pour COMMENTAIRES }
    begin
      XHG := 1;
      YHG := 16;
      XBD := 37;
      YBD := 23;
      SavBuf := Nil;
      Tracee := FALSE;
      Cadre := TRUE;
      Move (DoublTrait, Bord, 6);
      Titre := '';
      SavCoul := 7;
      CoulCad := CoulCadre;
      SavX := 0;
      SavY := 0;
    end;  { fenetre 5 }

  NumFenCour := 0;

{ illustration de la fenetre ECRAN }
  FenCharge (1);
  Cls;
  GoXY (0, 0);
  Chaine := '1234567890';
  for I := 1 to 191 do
    EcrisChn (Chaine);
  Chaine := '123456789';
  EcrisChn (Chaine);
  FenSauve;

  Read (Kbd, Ch);
{ illustration de la ligne MENU }
  FenCharge (4);
  Cls;
  GoXY (0, 0);
  Chaine := '                Ceci est la fenetre pour le MENU
HORIZONTAL';
  EcrisChn (Chaine);
  FenSauve;
```

```
   Read (Kbd, Ch);
{ mise -en -oeuvre et illustration de la fenetre MEMOIRE CENTRALE }
   FenCharge (2);
   Cls;
   GoXY (0, 0);
   Chaine := '              |             |             ';
   EcrisChnLn (Chaine);
   Chaine := ' 1234567890AB|1234567890AB|1234567890AB';
   for I := 1 to 3 do
     EcrisChnLn (Chaine);
   Chaine := '--------------+-------------+-------------';
   EcrisChn (Chaine);
   Chaine := ' 1234567890AB|1234567890AB|1234567890AB';
   for I := 1 to 3 do
     EcrisChnLn (Chaine);
   Chaine := '--------------+-------------+-------------';
   EcrisChn (Chaine);
   Chaine := ' Ceci est la fenetre "operations" ';
   for I := 1 to 12 do
     EcrisChnLn (Chaine);
   EcrisChn (Chaine);
   FenSauve;

   Read (Kbd, Ch);
{ illustration de la fenetre COMMENTAIRES }
   FenCharge (5);
   Cls;
   GoXY (0, 0);
   Chaine := 'Ceci est la fenetre pour les messages explicatifs. ';
   for I := 1 to 6 do
     EcrisChn (Chaine);
   FenSauve;

   Read (Kbd, Ch);
{ retour a la fenetre ECRAN }
   Fenetres [1].SavCoul := white;
   FenCharge (1);
   GoXY (18, 11);
   Chaine := ' Comme vous le constatez, ceci est l''écran ';
   EcrisChn (Chaine);
   Fenetres [1].SavCoul := lightgray;

   Read (Kbd, Ch);
{ retour a la fenetre MEMOIRE CENTRALE }
   FenCharge (2);
   GoXY(1,2);
   Chaine := '     ??      ';
   EcrisChn (Chaine);

   Read (Kbd, Ch);
{ retour a al fenetre ECRAN }
   FenCharge (1);

   Read (Kbd, Ch);
{ fin d'execution }
   Curseur (TRUE);
   ClrScr;
```

```
End;   { InitDecoupage }

{
PROGAMME PRINCIPAL :
}
BEGIN
   Monochrome;      { modifications "hard" en cas de materiel monochrome }
   InitDecoupage
END.
```

A.1.2 Deuxième sous-système : PRAIS02.PAS.

```
{ Listage de PRAIS02.PAS }
Program Prograis02;
{*************************************************************************}
{*                                                                       *}
{*                  PROGRAMME de Demonstration no.2                      *}
{*                                                                       *}
{*                        SECOND SOUS-SYSTEME                            *}
{*                        de l'outil PROGRAIS                            *}
{*                                                                       *}
{*              *** ETABLISSEMENT d'un ENVIRONNEMENT ***                 *}
{*        *** MEMORISATION du TEXTE du PROGRAMME a executer ***          *}
{*        *** GESTION de l'ORDRE d'EXECUTION des OPERATIONS ***          *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{*              AUTEUR : Anne de Baenst-Vandenbroucke                    *}
{*                                                                       *}
{*************************************************************************}

{
-------------------------------------------------------------------------

  SECOND SOUS-SYSTEME :
  --------------------
  Outre l'etablissement d'un environnement (cfr premier sous-systeme),
  ce second sous-systeme permet de tester
  - la memorisation interne du texte du programme a executer,
  - la visualisation du texte dans la sous-fenetre des operations,
  - une simulation de la gestion de l'ordre d'execution des operations.

  Ce second sous-systeme "UTILISE" les modules suivants:
  - Librairie de base,
  - Librairie de fenetrage,
  - gerant des messages
  ainsi que
  - controleur de communication,
  - editeur de memoire centrale,
  - gerant d'ecran.

OBJECTIF :
  Montrer comment le nom du programme est choisi, comment le texte du
  programme est edite dans la fenetre des operations apres memorisation
  et comment est gere l'ordre d'execution des operations.
```

```
-------------------------------------------------------------------------}

{$I Defgloba.lib}  { declarations globales pour les librairies }
{$I Defdemo.inc}   { declarations propres a l'application }
{$I Routines.lib}  { librairie de base }
{$I Winprimi.lib}  { librairie de fenetrage }
{$I Messages.inc}  { gestion des messages }
{$I Initdemo.inc}  { controle de communication }
{$I Visuamc.inc}   { edition Memoire Centrale et ecran}

{-------------------------}
procedure TestAffichageCode;
{-------------------------}
{<<<
  Simule la gestion de l'ordre d'execution des operations de la facon
  suivante:
  - lit une a une les lignes du texte (la ligne en cours de lecture est
    visualisee en video inverse) jusqu'a la fin du texte,
  - revient a la premiere ligne du texte,
  - repasse a la derniere ligne du texte,
  - termine par la deuxieme ligne du texte.
  Le passage d'une ligne a l'autre se fait par la frappe d'une touche
  quelconque.
>>>}
var
   I  : integer;
   Ch : char;

Begin

   read (kbd,Ch);
   for I := 1 to FinOpTab do
      begin
         NoLigne := NouvelleLigne (I);
         read (Kbd,Ch);
      end;
   NoLigne := NouvelleLigne (1);
   read (Kbd,Ch);
   NoLigne := NouvelleLigne (FinOpTab);
   read (Kbd,Ch);
   NoLigne := NouvelleLigne (2);
   read (Kbd,Ch);

end;  { TestAffichageCode }

{
PROGRAMME PRINCIPAL :
}
Begin

   Commencer;
   InitExec;    { effectue la mise-en-oeuvre }
   if not(erreur) then
      begin
         DebAffOp := AfficherOp(1);
         TestAffichageCode;
      end;
```

```
   Terminer;

end.
```

A.1.3 Troisième sous-système : PRAIS03.PAS et PRAIS04.PAS.

```
{ Listage de PRAIS03.PAS }
Program Prograis03;
{*************************************************************************}
{*                                                                       *}
{*                  PROGRAMME de Demonstration no.3                      *}
{*                                                                       *}
{*                       TROISIEME SOUS-SYTEME                           *}
{*                        de l'outil PROGRAIS                            *}
{*                                                                       *}
{*            *** CONSTRUCTION DE L' ANALYSEUR LEXICAL ***               *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{*              AUTEUR : Anne de Baenst-Vandenbroucke                    *}
{*                                                                       *}
{*************************************************************************}

{
-------------------------------------------------------------------------

  TROISIEME SOUS-SYSTEME :
  ----------------------
  Ce troisieme sous-systeme permet de tester
  - l'analyseur lexical de l'interpreteur de programme.
  La demonstration est faite dans le cadre de l'environnement developpe
dans
  les premiers sous-systemes.

  Ce troisieme sous-systeme "UTILISE" les modules suivants:
  - Librairie de base,
  - Librairie de fenetrage,
  - controleur de communication,
  - gerant de messages,
  - editeur de memoire centrale
  - gerant d'ecran
  ainsi que
  - analyseur lexical.

OBJECTIF :
  passer en revue les differents lexemes du programme.
  Ces lexemes et leur signification sont montres dans la case no.1
  de la partie "information" de la MEMOIRE CENTRALE.

-------------------------------------------------------------------------}

{$I Defgloba.lib}      { declarations globales pour modules librairies }
{$I Defdemo.inc}       { declarations pour application }
{$I Routines.lib}      { routines de base }
{$I Winprimi.lib}      { routines de fenetrage }
```

```
{$I Messages.inc}      { gestion des erreurs }
{$I Initdemo.inc}      { controle communication }
{$I VisuaMC.inc}       { edition Memoire Centrale et ecran}
{$I Scanner.inc}       { analyseur syntaxique }


{--------------------}
procedure TestScanner;
{--------------------}
{<<<
    Permet de tester si l'analyse lexicale se fait bien.
    Montre successivement la nature du lexeme lu.
>>>}
var
     Num  : integer;
     Ch   : char;
     Sym  : Symbole;
     Mess : alpha;

Begin
{<<< amorcage : }
   NoLigne := NouvelleLigne(1);
   NoPos := 1;
   LigneCour := OpTab [1] . Code + CR;
{<<< boucle : }
   while NoLigne <= FinOpTab do
      begin
         Biiip;
         Sym := SymScan (NoLigne,NoPos);
         case Sym  of
           finlig : begin
                       num       := succ (NoLigne);
                       NoLigne   := NouvelleLigne (num);
                       NoPos     := 1;
                       LigneCour := OpTab [NoLigne] . Code + CR;
                       GoXY (IdXNom [1], IdYNom [1]);
                       Mess := 'Chgt. Ligne ';
                       EcrisChn (Mess);
                       GoXY (IdXTy  [1], IdYTy  [1]);
                       EcrisChn (TyBlc);
                       GoXY (IdXVal [1], IdYVal [1]);
                       EcrisChn (NomBlc);
                    end;
           ident  : begin
                       Attribut  := CoulId;
                       GoXY (IdXNom [1], IdYNom [1]);
                       EcrisChn (Id);
                       GoXY (IdXTy  [1], IdYTy  [1]);
                       Mess := 'Ident. ';
                       EcrisChn (Mess);
                       GoXY (IdXVal [1], IdYVal [1]);
                       EcrisChn (NomBlc);
                    end;
           entcon : begin
                       Attribut := CoulId;
                       GoXY (IdXNom [1], IdYNom [1]);
                       EcrisChn (NomBlc);
                       GoXY (IdXTy  [1], IdYTy  [1]);
```

```
                    Mess := 'INTEGER';
                    EcrisChn (Mess);
                    GoXY (IdXVal [1], IdYVal [1]);
                    EcrisChn (Id);
                end;
        arraykw .. writekw : begin
                    Attribut  := CoulId;
                    GoXY (IdXNom [1], IdYNom [1]);
                    EcrisChn (Id);
                    GoXY (IdXTy  [1], IdYTy  [1]);
                    EcrisChn (TyBlc);
                    GoXY (IdXVal [1], IdYVal [1]);
                    Mess := 'Mot Reservè ';
                    EcrisChn (Mess);
                end;
        plus .. affect : begin
                    Attribut  := CoulId;
                    GoXY (IdXNom [1], IdYNom [1]);
                    EcrisChn (NomBlc);
                    GoXY (IdXTy  [1], IdYTy  [1]);
                    EcrisChn (TyBlc);
                    GoXY (IdXVal [1], IdYVal [1]);
                    Mess :=  ' Symb. spec.';
                    EcrisChn (Mess);
                end;
      end;   { case }
    end;  { while FinOpTab }

end;  { TestScanner }

{
PROGRAMME PRINCIPAL :
}
Begin

   Commencer;
   InitExec;
   if erreur then
      goto fini;
   DebAffOp := AfficherOp(1);
   Delay (1000);
   TestScanner;
 fini :
   Terminer;

End.


{ Listage de PRAIS04.PAS }
Program Prograis04;
{*************************************************************************}
{*                                                                       *}
{*                  PROGRAMME de Demonstration no.4                      *}
{*                                                                       *}
{*                      TROISIEME SOUS-SYSTEME                           *}
{*                       de l'outil PROGRAIS                             *}
{*                                                                       *}
```

```
{*             *** CONSTRUCTION DE L' ANALYSEUR LEXICAL ***              *}
{*                        ( deuxieme version )                           *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{*                 AUTEUR : Anne de Baenst-Vandenbroucke                 *}
{*                                                                       *}
{*************************************************************************}

{
-------------------------------------------------------------------------

  TROISIEME SOUS-SYSTEME (version 2):
  -----------------------------------
  Ce troisieme sous-systeme permet de tester
  - l'analyseur lexical de l'interpreteur de programme.
  La demonstration est faite dans le cadre de l'environnement developpe
dans
  les premiers sous-systemes.

  Ce troisieme sous-systeme "UTILISE" les modules suivants:
  - Librairie de base,
  - Librairie de fenetrage,
  - controleur de communication,
  - gerant de messages,
  - editeur de memoire centrale
  - gerant d'ecran
  ainsi que
  - analyseur lexical.

OBJECTIF :
  passer en revue les differents lexemes du programme.
  Ces lexemes et leur signification sont montres a tour de role dans
  les six cases de la partie "information" de la MEMOIRE CENTRALE.

-------------------------------------------------------------------------}


{$I Defgloba.lib}     { declarations globales pour modules librairies }
{$I Defdemo.inc}      { declarations pour application }
{$I Routines.lib}     { routines de base }
{$I Winprimi.lib}     { routines de fenetrage }
{$I Messages.inc}     { gestion des erreurs }
{$I Initdemo.inc}     { controle de communication }
{$I VisuaMC.inc}      { edition Memoire Centrale et ecran}
{$I Scanner.inc}      { analyseur lexical }

{--------------------}
procedure TestScanner2;
{--------------------}
{<<<
    Permet de tester si l'analyseur lexical fonctionne bien (version 2).
>>>}
var
     I    : integer;
     Num  : integer;
     Ch   : char;
     Sym  : Symbole;
```

```
     Mess : alpha;

Begin
{<<< amorcage : }
   NoLigne := NouvelleLigne(1);
   NoPos := 1;
   LigneCour := OpTab [1] . Code + CR;
   I := 1;
 { boucle : }
   while NoLigne <= FinOpTab do
      begin
         Biiip;
         Sym := SymScan (NoLigne,NoPos);
         case Sym  of
           finlig : begin
                       num       := succ (NoLigne);
                       NoLigne   := NouvelleLigne (num);
                       NoPos     := 1;
                       LigneCour := OpTab [NoLigne] . Code + #13;
                       GoXY (IdXNom [I], IdYNom [I]);
                       Mess := 'Chgt. Ligne ';
                       EcrisChn (Mess);
                       GoXY (IdXTy  [I], IdYTy  [I]);
                       EcrisChn (TyBlc);
                       GoXY (IdXVal [I], IdYVal [I]);
                       EcrisChn (NomBlc);
                    end;
           ident  : begin
                       Attribut := CoulId;
                       GoXY (IdXNom [I], IdYNom [I]);
                       EcrisChn (Id);
                       GoXY (IdXTy  [I], IdYTy  [I]);
                       Mess := 'Ident. ';
                       EcrisChn (Mess);
                       GoXY (IdXVal [I], IdYVal [I]);
                       EcrisChn (NomBlc);
                    end;
           entcon : begin
                       Attribut := CoulId;
                       GoXY (IdXNom [I], IdYNom [I]);
                       EcrisChn (NomBlc);
                       GoXY (IdXTy  [I], IdYTy  [I]);
                       Mess := 'INTEGER';
                       EcrisChn (Mess);
                       GoXY (IdXVal [I], IdYVal [I]);
                       EcrisChn (Id);
                    end;
           arraykw .. writekw : begin
                       Attribut := CoulId;
                       GoXY (IdXNom [I], IdYNom [I]);
                       EcrisChn (Id);
                       GoXY (IdXTy  [I], IdYTy  [I]);
                       EcrisChn (TyBlc);
                       GoXY (IdXVal [I], IdYVal [I]);
                       Mess := 'Mot Reservè ';
                       EcrisChn (Mess);
                    end;
           plus .. affect : begin
```

```
                        Attribut := CoulId;
                        GoXY (IdXNom [I], IdYNom [I]);
                        EcrisChn (NomBlc);
                        GoXY (IdXTy  [I], IdYTy  [I]);
                        EcrisChn (TyBlc);
                        GoXY (IdXVal [I], IdYVal [I]);
                        Mess :=  ' Symb. spec.';
                        EcrisChn (Mess);
                     end;
            end;   { case }
            I := succ (I);
            if I = 7 then
               I := 1;
         end;  { while FinOpTab }

end;  { TestScanner }


{
PROGRAMME PRINCIPAL :
}
BEGIN

   Commencer;
   InitExec;
   if erreur then
      goto fini;
   DebAffOp := AfficherOp(1);
   Delay (1000);
   TestScanner2;
 fini :
   Terminer;

END.
```

A.1.4 Quatrième sous-système : PRAIS05.PAS.

```
{ Listage de PRAIS05.PAS }
Program Prograis05;
{*************************************************************************}
{*                                                                       *}
{*                   PROGRAMME de Demonstration no.5                     *}
{*                                                                       *}
{*                        QUATRIEME SOUS-SYSTEME                         *}
{*                         de l'outil PROGRAIS                           *}
{*                                                                       *}
{*          *** CONSTRUCTION DE L' ANALYSEUR SYNTAXIQUE ***              *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{*              AUTEUR : Anne de Baenst-Vandenbroucke                    *}
{*                                                                       *}
{*************************************************************************}

{
```

```
------------------------------------------------------------------------

  QUATRIEME SOUS-SYSTEME :
  ----------------------
  Ce quatrieme sous-systeme permet de tester
  - l'analyseur syntaxique de l'interpreteur de programme.
  La demonstration est faite dans le cadre de l'environnement developpe
dans
  les premiers sous-systemes.

  Ce quatrieme sous-systeme "UTILISE" les modules suivants:
  - Librairie de base,
  - Librairie de fenetrage,
  - gerant de messages,
  - controleur de communication,
  - editeur de memoire centrale
  - analyseur lexical
  ainsi que
  - analyseur syntaxique.

OBJECTIF :
  Tester le bon fonctionnement de l'analyseur syntaxique en
  passant en revue les constructions syntaxiques du programme.
      Remarque :
      Ce quatrieme sous-systeme n'est pas prevu pour de long texte.
      Il ne gère pas les sorties d'ecran!

------------------------------------------------------------------------}

{$I Defgloba.lib}     { declarations globales pour modules librairies }
{$I Defdemo.inc}      { declarations pour application }
{$I Routines.lib}     { routines de base }
{$I Winprimi.lib}     { gestion de fenetres }
{$I Messages.inc}     { gestion des erreurs }
{$I Initdemo.inc}     { controle de communication }
{$I VisuaMC.inc}      { gestion edition Memoire Centrale et ecran}
{$I Scanner.inc}      { analyseur lexical }
{$I AnalOp.inc}       { analyseur syntaxique }


{
PROGRAMME PRINCIPAL :
}
BEGIN

   Commencer;
   InitExec;
   if erreur then
      goto fini;
   DebAffOp := afficherOp(1);
   Delay (1000);
   TestAnalyseur;
 fini :
   Terminer;

END.
```

A.1.5 Cinquième sous-système : PRAIS06.PAS.

```
{ Listage de PRAIS06.PAS }
Program Prograis06;
{*****************************************************************************}
{*                                                                           *}
{*                    PROGRAMME de Demonstration no.6                        *}
{*                                                                           *}
{*                         CINQUIEME SOUS-SYSTEME                            *}
{*                          de l'outil PROGRAIS                              *}
{*                                                                           *}
{*                 *** GESTION DES STRUCTURES INTERNES ***                   *}
{*                   *** EDITION MEMOIRE CENTRALE ***                        *}
{*                                                                           *}
{*****************************************************************************}
{*                                                                           *}
{*                AUTEUR : Anne de Baenst-Vandenbroucke                      *}
{*                                                                           *}
{*****************************************************************************}

{
-------------------------------------------------------------------------------

  CINQUIEME SOUS-SYSTEME :
  ----------------------
  Ce cinquieme sous-systeme permet de tester
  - la gestion des tables internes du modele general de l'ordinateur,
  - l' edition de la section visible de la memoire centrale.
  La demonstration est faite dans le cadre de l'environnement developpe
  dans les premiers sous-systemes.

  Ce quatrieme sous-systeme "UTILISE" les modules suivants:
  - Librairie de base,
  - Librairie de fenetrage,
  - gerant de messages,
  - controleur de communication,
  - editeur de memoire centrale,
ainsi que
  - gerant des tables internes.

OBJECTIF :
  Tester le bon fonctionnement du gerant des tables internes

-------------------------------------------------------------------------------}

{$I Defgloba.lib}     { declarations globales pour modules librairies }
{$I Defdemo.inc}      { declarations pour application }
{$I Routines.lib}     { routines de base }
{$I Winprimi.lib}     { routines de fenrtrage }
{$I Messages.inc}     { gerant des erreurs }
{$I Initdemo.inc}     { controleur de communication }
{$I VisuaMC.inc}      { edition Memoire Centrale et ecran}
{$I TablesMC.inc}     { gerant des tables internes }
```

```
{-----------------------------}
Procedure TestVisuaInformations;
{-----------------------------}
{<<<
    Montrer la gestion de l'edition des informations
    en considerant differents cas d'especes.
    ( table interne de 50 cases )
>>>}
var
     I      : byte;
     Ch     : Char;
     NomId  : alpha;
     TypId  : Typage;

Begin

   TypId := intty;
   for I := 1 to 50 do
      begin
         str (I:2, NomId);
         NomId := 'Nom' + NomId + TyBlc;
         if TypId = intty then
            TypId := boolty
         else
            TypId := intty;
         AjouterId (NomId, variable, TypId);
      end;
   AfficherId (1);      { cas d'exception : DedAffId = 0 }
   read (kbd,Ch);
   AfficherId (2);      { cas sans chgt. de page (1ere page - 1ere ligne) }
   read (kbd,Ch);
   AfficherId (6);      { cas sans chgt. de page (1ere page - 2eme ligne)}
   read (kbd,Ch);
   AfficherId (8);      { changement de page : cas 1 }
   read (kbd,Ch);
   AfficherId (13);     { changement de page : cas 2 }
   read (kbd,Ch);
   AfficherId (24);     { changement de page : cas 3 }
   read (kbd,Ch);
   AfficherId (22);     { cas sans changement de page (1ere ligne) }
   read (kbd,Ch);
   AfficherId (26);     { cas sans changement de page (2eme ligne) }
   read (kbd,Ch);
   AfficherId (20);     { changement de page en arriere }
   read (kbd,Ch);
   AfficherId (50);     { changement de page en fin de table virtuelle }
   read (kbd,Ch);

end;

{
PROGRAMME PRINCIPAL :
}
BEGIN

   Commencer;
   CreerFens;
   InitFens;
```

```
   if erreur then
      goto fini;
   Delay (1000);
   TestVisuaInformations;
 fini :
   Terminer;

END.
```

A.2 Codage de la version finale provisoire de PROGRAIS.

```
{ Listage de PROGRAIS.PAS }
Program Prais07;
{*************************************************************************}
{*                                                                       *}
{*                     PROGRAMME  FINAL  TEMPORAIRE                      *}
{*                                de                                     *}
{*                             PROGRAIS                                  *}
{*                                                                       *}
{*                  outil informatique annexe a la                       *}
{*        Methode d'Initiation aux Raisonnements de la Programmation     *}
{*                     de R. Lesuisse et A. Borsu                        *}
{*                                                                       *}
{*                              ecrit par                                *}
{*                       A. de BAENST-VANDENBROUCKE                      *}
{*                                                                       *}
{*     dans le cadre du memoire de licence et maitrise en informatique   *}
{*                             Aout 1991                                 *}
{*                                                                       *}
{*                                                                       *}
{*                  langage reconnu : DEMO PASCAL                        *}
{*               ( mini langage de convention Pascal )                   *}
{*                                                                       *}
{*************************************************************************}

{$I Defgloba.lib}     { declarations globales pour les librairies        }
{$I Defdemo.inc}      { declarations propres a PROGRAIS                  }
{$I Routines.lib}     { librairie de base .                              }
{$I Winprimi.lib}     { librairie de fenetrage                           }
{$I Messages.inc}     { gerant des messages                              }
{$I Initdemo.inc}     { controleur de communication                      }
{$I Tablesmc.inc}     { gerant des tables internes                       }
{$I Visuamc.inc}      { edition de la Memoire Centrale et de l'Ecran     }
{$I Scanner.inc}      { analyseur lexical                                }
{$I Execprog.inc}     { analyseur syntaxique et executeur de programme   }


{
PROGRAMME PRINCIPAL :
--------------------
}

BEGIN

   Commencer;
```

```
Debut:
{<<<
 Creation de l'environnement avec choix du programme à executer :
>>>}
  InitExec;
  if erreur then
     goto fini;
  DebAffOp := afficherOp(1);
  Delay (1000);
{<<<
 Execution du programme :
>>>}
  LanceExec;

fini :
  repeat
     TT := AttTouche;
  until (TT in ['O','o','N','n']);
  if (TT = 'O') or (TT = 'o') then
     goto debut;

  Terminer;

END.
```

A.3 Codage des modules de déclarations.

A.3.1 DEFGLOBA.LIB.

```
{ Listage de DEFGLOBA.LIB }
{*************************************************************************}
{*                                                                       *}
{*                      D E C L A R A T I O N S                          *}
{*                           relatives aux                               *}
{*                        L I B R A I R I E S                            *}
{*          ( Librairie de base et Librairie de fenetrage )              *}
{*                                                                       *}
{*                 fait partie du logiciel  PROGRAIS                     *}
{*                                                                       *}
{*************************************************************************}

{
CONTENU :
   Declarations relatives aux routines elementaires et de fenetrages.
   Ces declarations concernent entre autre
      - l'environnement hard,
      - la description d'un fichier physique quelconque,
      - les fenetres ( et leurs encadrements ),
      - des noms logiques pour les codes ASCII speciaux,
   ainsi que les types relatifs aux registres du microprocesseur.
}
```

```
{-------------------------------------------------------------------------}
{                          CONSTANTES GLOBALES                            }
{-------------------------------------------------------------------------}

{---------------------- CONSTANTES  non typees ---------------------------}
const
{----------------- codes pour les touches speciales ---------------------}
   F1    =  #59;
   F2    =  #60;
   F3    =  #61;
   F4    =  #62;
   F5    =  #63;
   F6    =  #64;
   F7    =  #65;
   F8    =  #66;
   F9    =  #67;
   F10   =  #68;

   FG    =  #75;
   FD    =  #77;
   FH    =  #72;
   FB    =  #80;

   Ins   =  #82;
   Del   =  #83;
   Home  =  #71;
   Fin   =  #79;
   PgUp  =  #73;
   PgDn  =  #81;
   Esc   =  #27;
   CR    =  #13;

{-------------------------- codes speciaux -------------------------------}
   BS    =  #8;
   LF    =  #10;
   FF    =  #12;
   NUL   =  #0;
   BEL   =  #7;

{------------------------ CONSTANTES  typees -----------------------------}
const
{------------------------ Environnement Hard -----------------------------}

 { !!! REMARQUE !!!: ces constantes dependent du materiel utilise }

   AdrVideo  : integer = $B800;  { Adresse Segment }
   CursOn1   : integer = $06;    { Curseur On }
   CursOn2   : integer = $07;
   CursOff1  : integer = $08;    { Curseur Off }
   CursOff2  : integer = $00;

   Neige     : boolean = FALSE;
 { !!! REMARQUE !!!: il faut de preference mettre  Neige = FALSE  }
 {   et mettre Neige = TRUE uniquement s'il y a neige a l'ecran   }
 {   ce qui est le cas des "Olivetti M24"                         }
```

```
{-------------------------- Fichier Physique ---------------------------}
   LgNomfich = 20;              { longueur max. pour un nom de fichier }


{------------------------------ Fenetres --------------------------------}
   MaxFen   = 10;               { Nombre maximum de fenetres }
   CoulTitre  : integer = 112; { fond grisclair + noir }
   CoulCadre  : integer = 15;  { blanc }
   CoulTxt    : integer = 14;  { jaune }
   CoulFond   : integer = 0;   { noir }


{---------------------- Encadrements des Fenetres ----------------------}
   SimplTrait : array [1..6] of char = (#218,#191,#192,#217,#179,#196);
   DoublTrait : array [1..6] of char = (#201,#187,#200,#188,#186,#205);
 { caracteres ASCII pour les encadrements en trait simple : }
   HautG      : char = #218;
   HautD      : char = #191;
   BasG       : char = #192;
   BasD       : char = #217;
   Verti      : char = #179;
   Horiz      : char = #196;


{------------------------------------------------------------------------}
{                            TYPES GLOBAUX                               }
{------------------------------------------------------------------------}

{--------------------- registres du microprocesseur ---------------------}
type

   Registres = record
                 AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : integer;
               end;


{---------------------- chaines de longueur variable --------------------}
   Str2     = string [2];
   Str78    = string [78];
   Str255   = string [255];  { string max. de TURBO PASCAL 3.0 }

   Nomfich  = string [LgNomFich];


{------------------------------ Fenetres --------------------------------}
   PtrBuf   = ^Buffer;
   Buffer   = array [0..2000] of integer;
 { caracteristiques generales d'une fenetre : }
   Window   = record
                XHG,
                YHG,
                XBD,
                YBD      : integer;                      { coordonnees }
                Bord     : array [1..6] of char; { type d'encadrement }
                Titre    : str78;                          { titre }
                SavBuf   : PtrBuf;                       { contenu }
                Tracee,                          { statut de tracage }
                Cadre    : boolean;         { statut de l'encadrement }
```

```
                SavX,
                SavY,           { position du curseur dans la fenetre }
                SavCoul  : integer;    { attribut video de la fenetre }
                CoulCad  : byte;    { attribut video de l'encadrement }
              end;


{-------------------------------------------------------------------------}
{                          VARIABLES GLOBALES                             }
{-------------------------------------------------------------------------}

var
{----------------------- Saisie d'un caractere --------------------------}
   TouchFonct : boolean;        { variable test pour touche de fonction }
   TT         : Char; { variable pour voir si un caractere a ete frappe }

{----------------------------- Ecran Absolu ------------------------------}
   X,
   Y          : integer;           { position courante sur ecran absolu }

{----------------------------- Fenetres ----------------------------------}
   Fenetres   : array [1..MaxFen] of Window;
                          { base de donnees des fenetres }

 { caracteristiques de la fenetre courante : }
   NumFenCour,    { numero de la fenetre courante }

   NbColMax,      { nombre de colonnes et lignes maximum }
   NbLigMax,      {      de la fenetre courante          }

   FenHG_X,
   FenHG_Y,
   FenBD_X,
   FenBD_Y,       { coordonnees de la fenetre courante   }

   XFen,
   YFen,          { position du curseur dans la fenetre courante }

   Attribut   : integer;
                   { attribut video de la fenetre courante }
```

A.3.2 DEFDEMO.INC.

```
{ Listage de DEFDEMO.INC }
{*************************************************************************}
{*                                                                       *}
{*                       D E C L A R A T I O N S                         *}
{*                            relatives a                                *}
{*                          l'outil PROGRAIS                             *}
{*                                                                       *}
{*************************************************************************}

{
CONTENU :
```

```
   Declarations propres a l'outil PROGRAIS (autres que librairies).
   Ces declarations concernent entre autres
      - le programme source
      - la table des operations permettant de gerer
                la MEMOIRE CENTRALE : PARTIE OPERATIONS
      - la table des identificateurs permettant de gerer
                la MEMOIRE CENTRALE : PARTIE INFORMATIONS
      - la syntaxe concrete du Langege DEMO PASCAL
      - l'edition de la MEMOIRE CENTRALE et de l'envirronement
              . coordonnées d'edition,
              . couleurs d'edition, ...
      - la gestion proprement dite.
}

{-------------------------------------------------------------------------}
{                              LABELS                                     }
{-------------------------------------------------------------------------}
label
   fini,       { pour la gestion des erreurs }
   Debut;      { pour recommencer une execution }

{-------------------------------------------------------------------------}
{                      CONSTANTES non typees                              }
{-------------------------------------------------------------------------}
const
{ pour le programme Source :}
   LgLigneTxt     = 127; { longueur d'une ligne du programme source }
                 { ! choisi en fonction du TURBO PASCAL ! }

{MEMOIRE CENTRALE : PARTIE OPERATIONS}
{ pour la table des operations :}
   LgLigneCode    = 38;  { nbr. max. caracts. dans ligne code }
   MaxLi          = 100; { nbr. max. d'elements dans table }
   NbrAffLi       = 13;  { nbr. lignes codes visibles }

{MEMOIRE CENTRALE : PARTIE INFORMATIONS}
{ pour table des identificateurs :}
   LgAr           = 8;   { longueur max. d'un identificateur de tableau }
   LgId           = 12;  { longueur max. d'un identificateur }
       { !!! REMARQUE !!!: LgAr = LgId - 4 places pour [**] }
   MaxId          = 500; { nbr. max. d'identificateurs }
   MaxArray       = 99;  { nbr. max d'elements dans un tableau }
   NbrAffId       = 6;   { nbr. identificateurs visibles }

{ pour le traitement :}
   NbrKw          = 20;  { nbr. de MOTS RESERVES ou IDENT. STANDARDS }
                                 { en  DEMO PASCAL }


{-------------------------------------------------------------------------}
{                               TYPES                                     }
{-------------------------------------------------------------------------}
type
   Caracteres = set of char;

{ pour programme Source :}
   LigneTxt    = string [LgLigneTxt];
```

```
{ pour table des operations :}
   LigneCode  = string [LgLigneCode];
   { caracteristiques d'un element }
   ElOpTab    = record
                   Code    : LigneCode;
                end;

{ pour table des identificateurs :}
   Alpha      = string [LgId];
   Objet      = (variable);
   Typage     = (pasty, intty, boolty);  { donne le type simple }
   Typset     = set of Typage;
   SortTy     = (simple, arrayty);       { donne le type structure }
   { caracteristiques d'un element }
   ElIdTab    = record
                   Nom      : Alpha;
                   Obj      : Objet;
                   Typ      : Typage;
                   STyp     : SortTy;
                   ExistVal : boolean;
                   case T : Typage of
                      pasty  : (valpas  : char);
                      intty  : (valint  : integer);
                      boolty : (valbool : boolean);
                end;

{ pour le traitement }
   Symbole = ( plus      ,moins     ,fois      ,egal      ,nonegal   ,
               ppetit    ,ppetiteg  ,pgrand    ,pgrandeg  ,gparen    ,
               dparen    ,gcroch    ,dcroch    ,point     ,virg      ,
               ptvirg    ,deuxpt    ,ptpt      ,affect    ,arraykw   ,
               beginkw   ,booleankw ,divkw     ,dokw      ,elsekw    ,
               endkw     ,falsekw   ,ifkw      ,integerkw ,modkw     ,
               notkw     ,ofkw      ,programkw ,readkw    ,thenkw    ,
               truekw    ,varkw     ,whilekw   ,writekw   ,ident     ,
               entcon    ,finlig  );
                     { symboles abstraits pour le DEMO PASCAL }

   Symset  = set of Symbole;


{-------------------------------------------------------------------------}
{                          CONSTANTES typees                              }
{-------------------------------------------------------------------------}
const
{ coordonnees du debut d'affichage des operations }
{ dans la fenetre MEMOIRE CENTRALE (fenetre 2) :  }
   OpYH    : byte   = 8;  { avec premiere ligne a 9 }
   OpXG    : byte   = 1;

{ coordonnees des cases pour identificateurs a l'ecran }
{ dans la fenetre MEMOIRE CENTRALE (fenetre 2) :       }
   IdXNom  : array [1..6] of byte = ( 1, 14, 27,  1, 14, 27);
   IdYNom  : array [1..6] of byte = ( 1,  1,  1,  5,  5,  5);
   IdXAr   : array [1..6] of byte = ( 1, 14, 27,  1, 14, 27);
   IdYAr   : array [1..6] of byte = ( 2,  2,  2,  6,  6,  6);
   IdXTy   : array [1..6] of byte = ( 6, 19, 32,  6, 19, 32);
   IdYTy   : array [1..6] of byte = ( 2,  2,  2,  6,  6,  6);
```

```
   IdXVal   : array [1..6] of byte = ( 1, 14, 27,  1, 14, 27);
   IdYVal   : array [1..6] of byte = ( 3,  3,  3,  7,  7,  7);

{ couleur dans la fenetre ECRAN (fenetre 1): }
   F1TxtCoul  : byte  = 14;  { jaune }

{ couleurs dans la fenetre MEMOIRE CENTRALE (fenetre 2) : }
   F2TxtCoul  : byte  = 15;  { blanc }
   F2CadCoul  : byte  = 15;  { blanc }
   CoulOp     : byte  = 14;  { jaune }
   InvCoulOp  : byte  = 96;  { fond brun + noir }
   CoulId     : byte  = 12;  { rouge clair }
   CoulAr     : byte  = 72;  { fond rouge + gris fonce }
   CoulVide   : byte  = 7;   { gris clair }

{ couleurs dans la fenetre ERREURS (fenetre 3) :}
   F3TxtCoul  : byte = 6;    { brun }
   F3CadCoul  : byte = 7;    { gris clair }

{ couleur dans la fenetre MENU (fenetre 4) : }
   F4TxtCoul  : byte = 64;   { fond rouge + noir }

{ couleurs dans la fenetre PARTIE DU PROGRAMME (fenetre 5) : }
   F5TxtCoul  : byte = 6;    { brun }
   F5CadCoul  : byte = 7;    { gris clair }

{ couleurs de la fenetre COMMENTAIRES (fenetre 6) : }
   F6TxtCoul  : byte = 6;    { brun }
   F6CadCoul  : byte = 7;    { gris clair }

{ couleurs de la fenetre de FIN d'execution (fenetre 7) : }
   F7TxtCoul  : byte = 6;    { brun }
   F7CadCoul  : byte = 7;    { gris clair }

{ couleurs de la fenetre de la SPECIFICATION du programme (fenetre 8) : }
   F8TxtCoul  : byte = 15;    { blanc }
   F8InvCoul  : byte = 112;   { fond gris clair + noir }
   F8CadCoul  : byte = 14;    { jaune }

{ pour gommer dans la fenetre MEMOIRE CENTRALE (fenetre 2) : }
   { 38 caracteres blancs }
   CodeBlc : LigneCode  =  '                                      ';
   { 12 caracteres blancs }
   NomBlc  : alpha      =  '            ';
   { 7 caracteres blancs }
   TyBlc   : alpha      =  '       ';
   { 5 caracteres blancs }
   ArBlc   : alpha      =  '     ';

{ pour la gestion de la fenetre MEMOIRE CENTRALE (fenetre 2) : }
   CaseVide: alpha =  '??     ';      { indicateur de case memoire vide }
   PasVal  : alpha =  '          ? '; { indicateur de valeur inconnue }

{ mots reserves et ident. standards de DEMO PASCAL
                                      par ordre alphabetique : }
   Kw    : array[1..Nbrkw] of alpha
       = ('ARRAY       ','BEGIN       ','BOOLEAN     ','DIV         ',
          'DO          ','ELSE        ','END         ','FALSE       ',
```

```
          'IF          ','INTEGER     ','MOD         ','NOT         ',
          'OF          ','PROGRAM     ','READ        ','THEN        ',
          'TRUE        ','VAR         ','WHILE       ','WRITE       ');

{ symboles pour mots reserves et ident. standards de DEMO PASCAL : }
   KwSym : array[1..Nbrkw] of symbole
       = (arraykw, beginkw, booleankw, divkw,
          dokw, elsekw, endkw, falsekw,
          ifkw, integerkw, modkw, notkw,
          ofkw, programkw, readkw,thenkw,
          truekw, varkw, whilekw, writekw);

{ ensembles pour les symboles initiaux dans DEMO PASCAL : }
   ConstDebSy : Symset = [plus, moins, entcon, ident];
   FacDebSy   : Symset = [entcon, ident, gparen, notkw];
   OpDebSy    : symset = [ident, readkw, writekw, beginkw, ifkw, whilekw];


{-------------------------------------------------------------------------}
{                              VARIABLES                                  }
{-------------------------------------------------------------------------}
var
   Erreur    : boolean;   { variable test pour sortie en cas d'erreur }

{ pour le programme Source : }
   Source    : text;
   NomSource : NomFich;

{ pour la table des operations : }
   OpTab : array [1..MaxLi] of ElOpTab;
   FinOpTab  : byte;
   DebAffOp  : integer;
 { !!! REMARQUE :
       DebAffOp est de type "integer" car possibilite de valeur negative }

{ pour la table des identificateurs : }
   IdTab : array [1..MaxId] of ElIdTab;
   FinIdTab  : byte;
   DebAffId   : integer;
 { !!! REMARQUE :
       DebAffId est de type "integer" car possibilite de valeur negative }

{ pour le traitement : }
   { symboles pour symboles speciaux de 1 caractere  }
   Ssp       : array[char] of symbole;
 { !!! REMARQUE : "Ssp" doit etre initialise en debut d'application
        Cf. procedure InitDemoPlus de DEFDEMO.INC}

   NomProg   : Alpha;   { identificateur de programme }
   Ch        : char;    { dernier caractere lu }
   Sym       : Symbole; { dernier symbole lu }
   Id        : Alpha;   { dernier identificateur ou mot reserve analyse }
   Entss     : integer; { dernier entier analyse }

   NoLigne,             { No de la ligne courante }
   NoPos : integer;     { position courante dans la ligne courante }
   LigneCour : Str78;  { texte de la ligne courante + <retour a la ligne> }
```

```
   DebDecl   : byte; { No du premier ident. dans une declaration multiple }

   NomIdent  : Alpha;   { identificateur en traitement }
   ValExpr   : integer; { valeur de l'expression en traitement }
   ChnExpr   : Alpha;   { chaine associee a ValExpr }
```

A.4 Codage des modules de primitives.

A.4.1 ROUTINES.LIB.

```
{ Listage de ROUTINES.LIB }
{*************************************************************************}
{*                                                                       *}
{*                Primitives pour PROGRAIS relatives aux                 *}
{*                                                                       *}
{*                            M O D U L E                                *}
{*                                                                       *}
{*                                                                       *}
{*                          L I B R A I R I E                            *}
{*                            d e   B A S E                              *}
{*                                                                       *}
{*************************************************************************}

{-------------------------------------------------------------------------}
{
  CONTENU :
      ROUTINES elementaires suivantes :
      - Touche     : Saisit un caractere sans attendre la frappe,
      - AttTouche  : Idem avec attente,
      - Curseur    : Curseur visible ou non (mode texte uniquement),
      - Ecris      : Ecrit un caractere a la position courante du curseur,
      - Lis        : Lit un caractere a la position courante du curseur,
      - Open       : teste l'existance d'un fichier physique et l'ouvre
                                                        s'il existe,
      - Upper      : convertit une chaine de caractere en majuscules,
      - Bip        : emet un bip sonore,
      - Biiip      : emet un bip sonore et provoque un temps d'attente,
      - Monochrome : modifie les parametres Hard pour un PC monochrome.
      - Commencer  : fait les initialisation "hard"
      - Terminer   : terminaison "hard"

  Les declarations concernant ces routines se trouve dans le fichier
                          DEFGLOBA.LIB.
}
{-------------------------------------------------------------------------}

{--------------------}
Function Touche : char;
{--------------------}
{<<<
  Retourne le caractere correspondant a une touche sans attendre la frappe
  d'une touche au clavier. La variable "TouchFonct" est mise a TRUE si le
  caractere est associe a une touche de fonction.
```

```
>>>}
var
      Ch    : char;

Begin

   TouchFonct := FALSE;
   if KeyPressed then
     begin
       Read (Kbd, Ch);
       if KeyPressed then
         begin
           Read (Kbd, Ch);
           TouchFonct := TRUE;
         end;
     end
   else
     Ch := Chr (0);
   Touche := Ch;

End;  { Touche }


{------------------------}
Function AttTouche : char;
{------------------------}
{<<<
  Idem que la fonction Touche avec attente de la frappe d'une touche
  au clavier.
>>>}
var
     Ch : char;

Begin

   repeat
      Ch := Touche
   until (Ch <> NUL);
   AttTouche := Ch;

End;  { AttTouche }


{--------------------------------}
Procedure Curseur (Vis : Boolean);
{--------------------------------}
{<<<
  Rend le curseur visible ou non a l'ecran.
>>>}
var
      Regs  : Registres;

Begin

   with Regs do
     begin
       AX := $01 Shl 8;
       if Vis then
```

```
        CX := CursOn1  Shl 8 + CursOn2
      else
        CX := CursOff1 Shl 8 + CursOff2;
      Intr ($10, Regs);
    end;

End;  { Curseur }


{------------------------}
Procedure Ecris (C : char);
{------------------------}
{<<<
  Ecrit un caractere a l'ecran a la position courante du curseur (X,Y)
  avec la couleur donnee par la variable "Attribut" en utilisant la
  Memoire Video (ecriture instantanee).
>>>}
var
     AdrEcr    : integer;
     PtrVideo  : ^integer;

Begin

   AdrEcr   := X Shl 1 + Y * 160;
   PtrVideo := Ptr (AdrVideo, AdrEcr);
   Inline ($06/
           $C4/$BE/PtrVideo/
           $BA/$DA/$03/
           $8A/$3E/Attribut/
           $8A/$9E/C/
           $2E/
           $A1/Neige/
           $D1/$D8/
           $73/$0A/
           { 1. }
           $EC/
           $D0/$D8/
           $72/$FB/
           { 2.}
           $EC/
           $D0/$D8/
           $73/$FB/
           $89/$D8/
           { 3. }
           $AB/
           $07 );

End;  { Ecris }


{-----------------}
Function Lis : char;
{-----------------}
{<<<
  Lit un caractere a la position courante (X,Y) du curseur et retourne
  sa valeur.
>>>}
var
```

```
      I   : integer;
      Ch  : char;

Begin

   Ch := AttTouche;
   if TouchFonct then
         Lis := NUL
   else
         Lis := Ch;

End;  { Lis }


{-------------------------------------------------}
Function Open(var fp : text;Nom : NomFich) : boolean;
{-------------------------------------------------}
{<<<
  Verifie l'existence d'un fichier "NomFich" de type "text" sur support
  externe et ouvre le fichier en cas d'existence.
  Retourne la valeur TRUE si le fichier existe et FALSE sinon.
>>>}
Begin

   assign(fp,Nom);
   {$I-}
   reset(fp);
   {$I+}
   if IOresult <> 0 then
      begin
         Open := false;
         close(fp);
      end
   else
      Open := true;

End;  { Open }


{-----------------------------}
Function Upper (Var C) : Str255;
{-----------------------------}
{<<<
  Convertit une chaine de caractere de longueur quelconque en majuscules.
>>>}
var
     Temp : Str255 absolute C;
     Tl   : Str255;
     I    : integer;

Begin

   Tl := Temp;
   for I := 1 to length (Tl) do
      Tl[I] := upcase (Tl[I]);
   Upper := Tl;

End;  { Upper }
```

```
{------------}
Procedure Bip;
{------------}
{<<<
  Emet un BIP sonore
>>>}
Begin

   write (BEL);

End; { Bip }


{--------------}
Procedure Biiip;
{--------------}
{<<<
  Emet un BIP sonore et provoque un temps d'attente.
>>>}
Begin

   write (BEL);
   delay (2000);      { read (kbd, ch); }

End; { Biiip }


{--------------------}
Procedure  Monochrome;
{--------------------}
{<<<
  Convertit les parametres d'environnement Hard predefinis
  en cas de PC Monochromme.
>>>}
Begin

   if Mem[$0000:$0449] = 7 then
      begin
         AdrVideo := $B000;
         CursOn1  := $0C;
         CursOn2  := $0D;
         CursOff1 := $0E;
         CursOff2 := $00;
      end;

End;

{-------------------------------- DEBUT --------------------------------}
{------------------}
procedure Commencer;
{------------------}
{<<<
  Fait les initialisations de depart ("hardware")
>>>}

Begin
```

```
   Monochrome;
   TextMode;
   TextBackGround (CoulFond);
   TextColor (CoulTxt);
   Curseur (FALSE);
   Erreur := FALSE;

End;

{-------------------------------- FIN ------------------------------------}
{-----------------}
Procedure Terminer;
{-----------------}
{<<<
  retour au dos (ou au turbo pascal)
>>>}

Begin

   Curseur (TRUE);
   gotoXY(1,1);
   clrscr;
   write(' C''EST FINI ');
   repeat until keypressed;

End;
```

A.4.2 WINPRIMI.LIB.

```
{ Listage de WINPRIMI.LIB }
{*****************************************************************************}
{*                                                                           *}
{*                Primitives pour PROGRAIS relatives aux                     *}
{*                                                                           *}
{*                              M O D U L E                                  *}
{*                                                                           *}
{*                                                                           *}
{*                          L I B R A I R I E                                *}
{*                               d e                                         *}
{*                          F E N E T R A G E                                *}
{*                                                                           *}
{*****************************************************************************}

{---------------------------------------------------------------------------}
{
CONTENU :
  Primitives relatives a la gestion des fenetres, a savoir
  - Sensscroll : gere le "scrolling" vertical dans la fenetre courante,
  - Scroll     : produit un "scrolling" vertical d'une ligne dans la
                 fenetre courante,
  - Cls        : efface la fenetre courante,
  - EcrisCar   : ecrit un caractere dans la fenetre courante avec
                 mise-a-jour du curseur (ecriture instantanee),
  - LisCar     : retourne le caractere lu dans la fenetre courante a
```

```
                la position courante du curseur avec mise a jour
                du curseur.
  - GoXY       : a le meme effet que la primitive "GotoXY" classique
                dans la fenetre courante,
  - EcrisChn   : ecrit une chaine de caracteres dans la fenetre courante
                avec mise-a-jour du curseur (ecriture instantanee),
  - EcrisChnLn : idem que "EcrisChn" avec fin de ligne,
  - LisChn     : lit une chaine de caracteres dans la fenetre courante
                a la position courante du curseur avec mise-a-jour
                du curseur.
  - TraceFen   : trace une fenetre a l'ecran,
  - FenSauve   : sauve les parametres de la fenetre courante,
  - FenCharge  : charge une fenetre.

  Une fenetre est definie par :
      - les coordonnees de reperage sur l'ecran (XHG,YHG,XBD,YBD),
      - un cadre eventuel (d'attribut video de couleur "CoulCad"),
      - un titre eventuel, en cas de cadre,
      - un attribut video de couleur SavCoul,
      - un buffer de sauvegarde du contenu et de ses attributs,

  Les modifications dans une fenetre sont possibles uniquement dans la
  fenetre de travail ou derniere fenetre chargee.Pour cette fenetre,
  il faut definir :
      - NbColMax   (Nombre de colonnes maximum),
      - NbLigMax   (Nombre de lignes maximum),
      - FenHG_X, FenHG_Y, FenBD_X, FenBD_Y (coordonnees courantes),
      - Attribut (attribut video courant).

  Les declarations concernant ces routines se trouve dans le fichier
                        DEFGLOBA.LIB.
}
{-------------------------------------------------------------------------}

{-----------------------------------------------}
Procedure SensScroll (NL :integer; Haut : Boolean);
{-----------------------------------------------}
{<<<
  Effectue le "scrolling" vertical de NL lignes dans la fenetre courante
              vers le haut si Haut = TRUE,
              vers le bas  si Haut = FALSE.
  Efface la fenetre dans le cas ou "NL" = 0.
>>>}
var
     Regs : Registres;

Begin

   with Regs do
     begin
       AX := 1792 + NL;
       CX := FenHG_Y Shl 8 + FenHG_X;
       DX := FenBD_Y Shl 8 + FenBD_X;
       BX := Attribut Shl 8;
       if Haut then
         AX := AX - 256;
     end;
   Intr ($10, Regs);
```

```
End; { SensScroll}


{-------------------------------}
Procedure Scroll (Haut : boolean);
{-------------------------------}
{<<<
  Scrolling vertical d'une ligne dans la fenetre courante.
>>>}
Begin

   SensScroll (1, Haut);

End;  { Scroll }


{------------}
Procedure Cls;
{------------}
{<<<
  Efface la fenetre courante et replace le curseur en (0,0)
>>>}
Begin

   SensScroll (0, False);
   XFen := 0;
   YFen := 0;

End;  { Cls }


{----------------------------}
Procedure EcrisCar (Ch : char);
{----------------------------}
{<<<
  Ecrit le caractere "Ch" dans la fenetre courante a la position courante
  du curseur avec mise a jour du curseur.
>>>}

Begin

   case Ch of
      BS  : begin
               if (Xfen <> 0) or (YFen <> 0) then
                  begin
                    if XFen = 0 then
                      begin
                        XFen := NbColMax - 1;
                        YFen := pred (YFen);
                      end
                    else
                      XFen := pred (XFen);
                    X := FenHG_X + XFen;
                    Y := FenHG_Y + YFen;
                    Ecris (' ');
                  end;
            end;  { BS }
```

```
      LF : YFen := succ (YFen);

      FF : Cls;

      CR : begin
               XFen := 0;
               YFen := succ (YFen);
           end; { CR }

      else
         begin
           X := FenHG_X + XFen;
           Y := FenHG_Y + YFen;
           Ecris (Ch);
           XFen := succ (Xfen);
         end;  { else }

   end; { case Ch }
   if XFen > NbColMax then
      begin
        XFen := 0;
        YFen := succ (YFen);
      end;
   if Yfen > NbLigMax then
      begin
        Scroll (TRUE);
        YFen := NbLigMax;
      end;

End;  { EcrisCar }


{---------------------}
Function LisCar : Char;
{---------------------}
{<<<
  Retourne le caractere lu dans la fenetre courante a la position courante
  du curseur avec mise a jour du curseur.
>>>}
var
   Ch  : char;

Begin

   X := FenHG_X + XFen;
   Y := FenHG_Y + YFen;
   Ch := Lis;
   case Ch of
      BS  : begin
               if (Xfen <> 0) or (YFen <> 0) then
                  begin
                    if XFen = 0 then
                      begin
                        XFen := NbColMax - 1;
                        YFen := pred (YFen);
                      end
                    else
```

```
                    XFen := pred (XFen);
                  X := FenHG_X + XFen;
                  Y := FenHG_Y + YFen;
                  LisCar := BS;
                  Ecris (' ');
                end;
          end;  { BS }

      CR : begin
              LisCar := CR;
          end; { CR }

      else
         begin
           X := FenHG_X + XFen;
           Y := FenHG_Y + YFen;
           LisCar := Ch;
           Ecris (Ch);
           XFen := succ (Xfen);
        end;  { else }

   end; { case Ch }
   if XFen > NbColMax then
      begin
        XFen := 0;
        YFen := succ (YFen);
      end;
   if Yfen > NbLigMax then
      begin
        Scroll (TRUE);
        YFen := NbLigMax;
      end;

End;  { LisCar }


{-------------------------------------------------}
Procedure FenReadLn ( XPos, YPos : integer; var C);
{-------------------------------------------------}
{<<<
  a le meme effet que la combinaison
             "gotoxy" et "readln"
  dans la fenetre courante.
>>>}
var
   Str   : str255 absolute C;

Begin

   GotoXY (FenHG_X + XPos + 1, FenHG_Y + Ypos + 1);
   readln (Str);

End;   { CapteChn }


{-----------------------------------}
Procedure GoXY (XPos, YPos : integer);
{-----------------------------------}
```

```
{<<<
  a le meme effet que la primitive "gotoXY" classique dans la fenetre
  courante.
>>>}
Begin

   if (XPos >= 0) and (Xpos <= NbColMax) then
      if (YPos >= 0) and (Ypos <= NbLigMax) then
         begin
           X := FenHG_X + XPos;
           Y := FenHG_Y + YPos;
           XFen := XPos;
           YFen := YPos;
         end;

End;  { GoXY }


{------------------------}
Procedure EcrisChn (var C);
{------------------------}
{<<<
  Ecrit une chaine de caracteres dans la fenetre courante a la position
  courante du curseur avec mise-a-jour du curseur.
>>>}
var
      Str  : str255 absolute C;
      I    : integer;

Begin

   for I := 1 to length (Str) do
      EcrisCar (Str[I]);

End;  { EcrisChn }


{--------------------------}
Procedure EcrisChnLn (var C);
{--------------------------}
{<<<
  Idem que la primitive "EcrisChn" avec, en plus, fin de ligne.
>>>}
var
      Str  : str255 absolute C;
      I    : integer;

Begin

      EcrisChn (Str);
      EcrisCar (CR);

End;  { EcrisChnLn }


{----------------------}
Procedure LisChn (var C);
{----------------------}
```

```
{<<<
  Lit une chaine de caracteres dans la fenetre courante a la position
  courante du curseur avec mise-a-jour du curseur.
>>>}
var
      Str  : str255 absolute C;
      I    : integer;
      Ch   : Char;

Begin

   I := 1;
   Ch := LisCar;
   while Ch <> CR do
      begin
         if Ch = BS then
            begin
               if I <> 1 then
                     I := I - 1;
            end
         else
            begin
               Str[I] := Ch;
               I := I + 1;
            end;
         Ch := Liscar;
      end;
   Str[0] := chr( I - 1);

End;  { LisChn }
{-----------------------------------}
Procedure TraceFen (Numero : integer);
{-----------------------------------}
{<<<
  Trace la fenetre "Numero".
>>>}
var
      I,
      J,
      K   : integer;

Begin

   with Fenetres[Numero] do
     begin
       Attribut := SavCoul;
       Cls;
       XFen := 0;
       YFen := 0;
       if Length (Titre) > NbColMax -2 then
         Titre [0] := Chr (NbColMax - 2);
       if Cadre then
         Begin
           Move (Bord, HautG, 6);
           Attribut := CoulCad;
           EcrisCar (HautG);
           if Length (Titre) <> 0 then
             begin
```

```
            I := ( NbColMax - Length (Titre) ) Shr 1;
            Attribut := CoulTitre;
            while XFen < I do
               EcrisCar (' ');
            EcrisChn (Titre);
            while XFen < NbColMax do
               EcrisCar (' ');
            Attribut := CoulCad;
          end
        else
          for I:= 1 to NbColMax - 1 do
              EcrisCar (Horiz);
        EcrisCar (HautD);
        for I := 1 to NbLigMax - 1 do
          begin
            XFen := 0;
            YFen := I;
            EcrisCar (Verti);
            XFen := NbColMax;
            EcrisCar (Verti);
          end;
        XFen := 0;
        YFen := NbLigMax;
        EcrisCar (BasG);
        for I := 1 to NbColMax - 1 do
            EcrisCar  (Horiz);
        X := XBD;
        Y := YBD;
        Ecris (BasD);
      end  { if cadre }
    else
      begin
        if Length (Titre) <> 0 then
          begin
            I := ( NbColMax - Length (Titre) ) Shr 1;
            Attribut := CoulTitre;
            while XFen < I do
               EcrisCar (' ');
            EcrisChn (Titre);
            while YFen < 1 do
               EcrisCar (' ');
            Attribut :=  CoulCad;
          end;
      end;  { else cadre }
    Tracee := TRUE;
    XFen := 0;
    YFen := 0;
    Attribut := SavCoul;
  end;  { with fenetre}

End;  { TraceFen }


{-----------------}
Procedure FenSauve;
{-----------------}
{<<<
  Sauve les parametres de la fenetre courante.
```

```
>>>}
var
      Adeb,
      CL,
      CC,
      Compteur,
      Offset    : integer;
      PtrVideo,
      SBuf      : PtrBuf;

Begin

   if NumFenCour in [1..MaxFen] then
     begin
       with Fenetres [NumFenCour] do
         begin
           if SavBuf = Nil then
             New (SavBuf);
           SBuf := SavBuf;
           Offset := 160;
           Adeb := YHG * 160 + XHG Shl 1;
           PtrVideo := Ptr (AdrVideo, Adeb);
           CC := XBD - XHG + 1;
           CL := YBD - YHG + 1;
           Compteur := CC * CL;
           Inline($FC/
                  $1E/
                  $C5/$B6/PtrVideo/
                  $C4/$BE/SBuf/
                  $BA/$DA/$03/
                  $8B/$9E/CL/
                  { 1. }
                  $56/
                  $8B/$8E/CC/
                  $2E/
                  $A1/Neige/
                  $D1/$D8/
                  $73/$0A/
                  { 2. }
                  $EC/
                  $D0/$D8/
                  $72/$FB/
                  { 3. }
                  $EC/
                  $D0/$D8/
                  $73/$FB/
                  $A5/
                  $49/
                  $75/$EA/
                  $5E/
                  $81/$C6/$A0/$0/
                  $4B/
                  $75/$DD/
                  $1F);
           SavX := XFen;
           SavY := YFen;
           SavCoul := Attribut;
         end;  { with Fenetres }
```

```
    end;   { if NumFenCour }

End;  { FenSauve }


{-------------------------------------}
Procedure FenCharge (Numero : integer);
{-------------------------------------}
{<<<
  Charge la fenetre "Numero" comme fenetre courante.
>>>}
var
      Adeb,
      CL,
      CC,
      Compteur,
      Offset    : integer;
      PtrVideo,
      SBuf      : PtrBuf;

Begin

   if Numero <> NumFenCour then
     begin
       FenSauve;
       NumFenCour := Numero;
       with Fenetres [Numero] do
         begin
           FenHG_X := XHG;
           FenHG_Y := YHG;
           FenBD_X := XBD;
           FenBD_Y := YBD;
           NbColMax := FenBD_X - FenHG_X;
           NbLigMax := FenBD_Y - FenHG_Y;
           if not tracee then
             TraceFen (Numero);
           if Cadre then
             begin
               FenHG_X := succ (FenHG_X);
               FenHG_Y := succ (FenHG_Y);
               FenBD_X := pred (FenBD_X);
               FenBD_Y := pred (FenBD_Y);
               NbColMax := NbColMax - 2;
               NbLigMax := NbLigMax - 2;
             end
           else
             if Length (Titre) <> 0 then
               begin
                 FenHG_Y := succ (FenHG_Y);
                 NbLigMax := NbLigMax - 1;
               end;
           XFen := SavX;
           YFen := SavY;
           Attribut := SavCoul;
           if SavBuf <> Nil then
             begin
               SBuf := SavBuf;
               Offset := 160;
```

```
                Adeb := YHG * 160 + XHG Shl 1;
                PtrVideo := Ptr (AdrVideo, Adeb);
                CC := XBD - XHG + 1;
                CL := YBD - YHG + 1;
                Compteur := CC * CL;
                Inline($FC/
                       $1E/
                       $C5/$B6/SBuf/
                       $C4/$BE/PtrVideo/
                       $BA/$DA/$03/
                       $8B/$9E/CL/
                       { 1. }
                       $57/
                       $8B/$8E/CC/
                       $2E/
                       $A1/Neige/
                       $D1/$D8/
                       $73/$0A/
                       { 2. }
                       $EC/
                       $D0/$D8/
                       $72/$FB/
                       { 3. }
                       $EC/
                       $D0/$D8/
                       $73/$FB/
                       $A5/
                       $49/
                       $75/$EA/
                       $5F/
                       $81/$C7/$A0/$0/
                       $4B/
                       $75/$DD/
                       $1F);
              end;   { if SavBuf <> Nil }
            end;  { with Fenetres }
          end;  { if Numero <> NumFenCour }

End;  { FenCharge }
```

A.4.3 MESSAGES.INC.

```
{ Listage de MESSAGES.INC }
{*************************************************************************}
{*                                                                       *}
{*                                                                       *}
{*              Primitives pour PROGRAIS relatives aux                   *}
{*                                                                       *}
{*                          M O D U L E                                  *}
{*                                                                       *}
{*                                                                       *}
{*               G E R A N T   des  M E S S A G E S                      *}
{*                                                                       *}
{*************************************************************************}
```

```
{
CONTENU
   Procedures :
         - NoErreur
         - NoPart
         - NoComm
         - NoFin
         - NoOpt

FONCTION
   gerer tous les messages explicatifs et leur fenetrage
   Pour illustration, ont ete retenus
        - des messages gérant les erreurs
        - des messages indiquant la partie du programme en traitement
        - des messages de commentaires sur l'evolution de l'execution
        - le message de fin d'exécution
        - des messages de menu
}

{--------------------------------}
Procedure NoErreur(NumErr : byte);
{--------------------------------}
{<<<
    affiche le message d'erreur NumErr dans fenetre 3
    et met Erreur a "true"
>>>}
Const
     MessErr : array[1..5] of Str78
       =  (' Programme non trouvé ',
           ' Programme avec lignes de plus  de 38 caractères',
           ' Programme non compatible avec  DEMO PASCAL ',
           ' Programme vide ',
           ' Opération pas encore prise en  considération !');

var
    Message : Str78;

Begin

   Fencharge(3);
   GoXY (0, 1);
   Str(NumErr,Message);
   Message := ' ERREUR ' + Message + ' :';
   EcrisChnLn (Message);
   Message := MessErr [NumErr];
   EcrisChn (Message);
   GoXY (0, 4);
   Message := ' Autre execution ? (OUI/NON)';
   EcrisChn (Message);
   {Attribut := 0;}
   Erreur := true;

End;


{-----------------------------}
Procedure NoPart(NumInf : byte);
{-----------------------------}
```

```
{<<<
    affiche le message indiquant la partie du programme en traitement
    dans la fenetre 5
>>>}
Const
     MessInf : array[1..4] of Str78
       =  (' en-tête du programme',
           ' déclarations des variables',
           ' exécution des opérations',
           ' exécution finie !              FRAPPEZ UNE TOUCHE !');

var
    Message : Str78;
    DerFen  : byte;

Begin

   DerFen := NumFenCour;
   Fencharge(5);
   Cls;
   GoXY (0, 1);
   Message := MessInf [NumInf];
   EcrisChn (Message);
   Fencharge(DerFen);

End;


{-----------------------------}
Procedure NoComm(NumInf : byte);
{-----------------------------}
{<<<
    affiche les messages de commentaires dans fenetre 6
>>>}
Const
     MessInf : array[1..7] of Str78
       =  ('',
           ' Le nom de la variable est      mémorisé',
           ' Le type de la variable est     mémorisé',
           ' Cette variable est un tableau  ses éléments sont mémorisés',
           '    Opération d''affectation        en cours.',
           '    Opération de lecture            en cours.',
           '    Opération d''écriture           en cours.');

var
    Message : Str78;
    DerFen  : byte;

Begin

   DerFen := NumFenCour;
   Fencharge(6);
   Cls;
   GoXY (0, 1);
   Message := MessInf [NumInf];
   EcrisChn (Message);
   Fencharge(DerFen);
```

```
End;


{--------------}
Procedure NoFin;
{--------------}
{<<<
    affiche le message de fin d'execution dans fenetre 7
>>>}
var
     Message  : Str78;

Begin

   Fencharge(7);
   GoXY (0, 1);
   Message := '  Autre execution ? (OUI/NON)';
   EcrisChn (Message);

End;


{----------------------------}
Procedure NoOpt(NumInf : byte);
{----------------------------}
        { affiche les messages dans la fenetre 4 }
Const
     MessInf : array[1..2] of Str78
     = ('   <ESC> permet de parcourir les cases "INFORMATIONS"      ',
        ''#24' et '#25' : défilement haut et bas - <ESC> : Retour
Exécution');

var
    Message : Str78;
    DerFen  : byte;

Begin

   DerFen := NumFenCour;
   Fencharge(4);
   Cls;
   GoXY (10, 0);
   Message := MessInf [NumInf];
   EcrisChn (Message);
   Fencharge(DerFen);

End;
```

A.4.4 INITDEMO.INC.

```
{ Listage de INITDEMO.INC }
{*************************************************************************}
{*                                                                       *}
{*               Primitives pour PROGRAIS relatives aux                  *}
{*                                                                       *}
```

```
{*                              M O D U L E S                                 *}
{*                                                                            *}
{*                         CONTROLEUR DE COMMUNICATION                        *}
{*                         GERANT DES  TABLES INTERNES                        *}
{*                          INTERPRETEUR DE PROGRAMME                         *}
{*                                                                            *}
{*                     intervennant en DEBUT D'APPLICATION                    *}
{*                                                                            *}
{****************************************************************************}

{
CONTENU :
   Differentes procedures intervenant en debut d'application:
   - Procedures relatives au controleur de communication :
         - CreerFens
         - Enviro
         - InitFens
   - Procedures relatives au garnissage de la table des operations
         - OuvrirSource
         - SourceDansOpTab
   - Procedure relatives a la definition du langage DEMO PASCAL
         - InitDemoPlus
   - Procedure de lancement d'application
         - InitExec

FONCTION :
   Faire tout ce qui est necessaire pour le controle de
   l'environnement de PROGRAIS
    c.-a-d.
      Creer et editer les fenetres :
         - MEMOIRE CENTRALE
         - ECRAN
         - fenetres de messages :
            . MENU
            . nom du programme source
            . ERREURS
            . COMMENTAIRES
            . PARTIE DU PROGRAMME
            . nouvelle execution

   Faire le garnissage de la table des operations
      Ouvrir le fichier du programme a executer
      Mettre le fichier source dans MEMOIRE CENTRALE : PARTIE OPERATION


   donner les definitions pour le langage de programmation (DEMO PASCAL)
   qui ne peuvent pas etre definies par des déclarations.

   Lancer l'application
}


{------------------ MISE EN OEUVRE DU FENETRAGE -------------------------}

{------------------}
Procedure CreerFens;
{------------------}
{<<<
```

```
  mise-en-oeuvre le fenetrage (8 fenetres)
     fenetres 1 et 2 pour representer le Modele General de l'Ordinateur
     autres fenetres pour la gestion des messages
>>>}

Begin

   with Fenetres [1] do  { fenetre 1 : representation de l'ECRAN }
     begin
       XHG := 0;
       YHG := 0;
       XBD := 79;
       YBD := 23;
       SavBuf := NIL;
       Tracee := FALSE;
       Cadre  := FALSE;
       Move (SimplTrait, Bord, 6);
       Titre := '';
       SavCoul := F1TxtCoul;
       CoulCad := CoulCadre;
       SavX := 0;
       SavY := 0;
     end;  { fenetre 1 }

   with Fenetres [2] do { fenetre 2 : representation de la MEM CENTRALE }
     begin
       XHG := 38;
       YHG := 0;
       XBD := 79;
       YBD := 23;
       SavBuf := NIL;
       Tracee := FALSE;
       Cadre  := TRUE;
       Move (SimplTrait, Bord, 6);
       Titre := 'MEMOIRE CENTRALE';
       SavCoul := F2TxtCoul;
       CoulCad := F2CadCoul;
       SavX := 0;
       SavY := 0;
     end;  { fenetre 2 }

   with Fenetres [3] do  { fenetre 3 : gestion des ERREURS }
     begin
       XHG := 24;
       YHG := 15;
       XBD := 56;
       YBD := 21;
       SavBuf := NIL;
       Tracee := FALSE;
       Cadre  := TRUE;
       Move (DoublTrait, Bord, 6);
       Titre := 'ERREUR';
       SavCoul := F3TxtCoul;
       CoulCad := F3CadCoul;
       SavX := 0;
       SavY := 0;
     end;  { fenetre 3 }
```

```
with Fenetres [4] do  { fenetre 4 : gestion du MENU ( ligne 25 ) }
  begin
    XHG := 0;
    YHG := 24;
    XBD := 79;
    YBD := 25;
    SavBuf := NIL;
    Tracee := FALSE;
    Cadre  := FALSE;
    Move (SimplTrait, Bord, 6);
    Titre := '';
    SavCoul := F4TxtCoul;
    CoulCad := CoulCadre;
    SavX := 0;
    SavY := 0;
  end;  { fenetre 4 }

with Fenetres [5] do  { fenetre 5 : indication PARTIE DU PROGRAMME }
  begin
    XHG := 1;
    YHG := 19;
    XBD := XHG + 32;
    YBD := YHG + 4;
    SavBuf := NIL;
    Tracee := FALSE;
    Cadre  := TRUE;
    Move (DoublTrait, Bord, 6);
    Titre := 'PARTIE DU PROGAMME';
    SavCoul := F5TxtCoul;
    CoulCad := F5CadCoul;
    SavX := 0;
    SavY := 0;
  end;  { fenetre 5 }

with Fenetres [6] do  { fenetre 6 : gestion des COMMENTAIRES }
  begin
    XHG := 1;
    YHG := 13;
    XBD := XHG + 32;
    YBD := YHG + 4;
    SavBuf := NIL;
    Tracee := FALSE;
    Cadre  := TRUE;
    Move (DoublTrait, Bord, 6);
    Titre := 'COMMENTAIRES';
    SavCoul := F6TxtCoul;
    CoulCad := F6CadCoul;
    SavX := 0;
    SavY := 0;
  end;  { fenetre 6 }

with Fenetres [7] do  { fenetre 7 : gestion de la FIN de l'execution }
  begin
    XHG := 24;
    YHG := 16;
    XBD := 56;
    YBD := 20;
    SavBuf := NIL;
```

```
        Tracee := FALSE;
        Cadre  := TRUE;
        Move (DoublTrait, Bord, 6);
        Titre := '';
        SavCoul := F7TxtCoul;
        CoulCad := F7CadCoul;
        SavX := 0;
        SavY := 0;
      end;  { fenetre 7 }

   with Fenetres [8] do  { gestion du CHOIX du PROGRAMME }
     begin
       XHG := 15;
       YHG := 14;
       XBD := 65;
       YBD := 18;
       SavBuf := NIL;
       Tracee := FALSE;
       Cadre  := TRUE;
       Move (DoublTrait, Bord, 6);
       Titre := '';
       SavCoul := F8TxtCoul;
       CoulCad := F8CadCoul;
       SavX := 0;
       SavY := 0;
     end;  { fenetre 8 }

     NumFenCour := 0;
     ClrScr;  { place le curseur en (0,0) sur l'ecran }

End;  { CreerFens }


{---------------}
Procedure Enviro;
{---------------}
{<<<
  Mise-en oeuvre de la visualisation conforme a PROGRAIS.
>>>}

Begin

   FenCharge (1);
   FenCharge (5);
   FenCharge (6);
   FenCharge (2);

End;


{-----------------}
Procedure InitFens;
{-----------------}
var
   Ch      : Char;
   Chaine  : str78;
   I       : integer;
```

```
Begin
{ >>> chargement des fenetres : }
   Enviro;
   NoOpt (1); { fenetre 4 avec message "NoOpt" 1 }
   Cls;

{ >>> configuration de la MEMOIRE CENTALE :}
   GoXY (0, 0);
   Chaine := '              |             |             ';
   for I := 1 to 4 do
      EcrisChn (Chaine);
   Chaine := '--------------+-------------+-------------';
   EcrisChn (Chaine);
   Chaine := '              |             |             ';
   for I := 1 to 3 do
      EcrisChn (Chaine);
   Chaine := '--------------+-------------+-------------';
   EcrisChn (Chaine);
   Chaine := '??';
   Attribut := CoulVide;
   for I := 1 to 6 do
      begin
         GoXY(IdXTy[I],IdYTy[I]);
         EcrisChn (Chaine);
      end;
   GoXY (0, 0);
   Attribut := CoulId;   FenSauve;

End;  { InitFens }
{ !!! REMARQUE :
        : en fin de InitFens NumFenCour = 2 }


{---------------- INITIALISATIONS de la TABLE des OPERATIONS -------------}

{---------------------}
Procedure OuvrirSource;
{---------------------}
{<<<
  Gestion du choix du programme a executer
>>>}
var
   Chaine : str78;

Begin

   FenCharge (8);
   Cls;
   GoXY (0, 0);
   Chaine := ' ENTREZ le NOM du PROGRAMME à EXECUTER : ';
   EcrisChnLn (Chaine);
   Attribut := F8InvCoul;
   GoXY (1, 1);
   EcrisChn (CodeBlc);
   GoXY (2, 1);
   LisChn (NomSource);
   NomSource := Upper(NomSource);
   if ( (pos('.', NomSource) = 0) and (pos('.PAS', NomSource) = 0)) then
```

```
        NomSource := Nomsource + '.PAS';
   while Not Open(Source, NomSource) do
      begin
         Attribut := F8TxtCoul;
         Cls;
         GoXY (0, 0);
         Chaine := ' Le PROGRAMME ';
         EcrisChn (Chaine);
         Attribut := F8InvCoul;
         Chaine := Nomsource;
         EcrisChn (Chaine);
         Attribut := F8TxtCoul;
         Chaine := ' est INNEXISTANT ';
         EcrisChnLn (Chaine);
         Chaine := ' ENTREZ un NOUVEAU NOM de PROGRAMME :';
         EcrisChnLn (Chaine);
         Attribut := F8InvCoul;
         GoXY (1, 2);
         EcrisChn (CodeBlc);
         GoXY (2, 2);
         LisChn (Nomsource);
         NomSource := Upper(NomSource);
         if ( (pos('.', NomSource) = 0) and (pos('.PAS', NomSource) = 0))
            then NomSource := Nomsource + '.PAS';
      end;

   Enviro;
   GoXY (0,0);

end;  { OuvrirSource }


{------------------------}
Procedure SourceDansOpTab;
{------------------------}
           { CETTE PROCEDURE LIMITE LE TRAITEMENT A UN CODE SOURCE DONT LES
             LIGNES ONT AU PLUS 38 CARACTERES ! }
{<<<
   introduction du texte du programme a executer dans
   la MEMOIRE CENTRALE : PARTIE OPERATION
>>>}

var
     Nocour    :byte;
     Tampontxt : Lignetxt;

begin

   Nocour   := 0;
   while not eof(Source) do
      begin
         Nocour := Succ(Nocour);
         readln(Source,Tampontxt);
         if length(Tampontxt) > LgLigneCode then
            begin
               NoErreur(2); {programmes de plus de 38 caracteres}
               Exit;
            end;
```

```
         with OpTab[Nocour] do
            Code := Tampontxt;
      end;
      close(Source);
      FinOpTab := Nocour;
      DebAffOp := 0;

end;  { SourceDansOpTab }


{-------------------- INITIALISATIONS pour LANGAGE ----------------------}

{--------------------}
procedure InitDemoPlus;
{--------------------}

Begin

{ >>>
   Definitions des caracteres speciaux de DEMO PASCAL : }
   Ssp['+']  := plus       ; Ssp['-']  := moins;
   Ssp['*']  := fois       ; Ssp['=']  := egal;
   Ssp['<']  := ppetit     ; Ssp['>']  := pgrand;
   Ssp['(']  := gparen     ; Ssp[')']  := dparen;
   Ssp['[']  := gcroch     ; Ssp[']']  := dcroch;
   Ssp['.']  := point      ; Ssp[',']  := virg;
   Ssp[';']  := ptvirg     ; Ssp[':']  := deuxpt;

   NoLigne := 0;  { pas de ligne en traitement }

   FinIdTab :=  0; { table des IDENTIFICATEURS vide }
   DebAffId :=  0; { table des IDENTIFICATEURS pas affichee }

End;


{---------------- INITIALISATIONS pour APPLICATION ----------------------}

{-----------------}
Procedure InitExec;
{-----------------}

Begin

   CreerFens;
   InitFens;
   OuvrirSource;
   SourceDansOpTab;
     { possibilite d'erreur no. 2 }
   if Erreur then
      Exit;
   if FinOpTab <> 0 then
      InitDemoPlus
   else
      NoErreur (4); {programme vide}
End; { InitExec }
```

A.4.5 TABLESMC.INC.

```
{ Listage de TABLESMC.INC }
{*************************************************************************}
{*                                                                       *}
{*              Primitives pour PROGRAIS relatives aux                   *}
{*                                                                       *}
{*                          M O D U L E                                  *}
{*                                                                       *}
{*          G E R A N T  des T A B L E S  I N T E R N E S                *}
{*                                                                       *}
{*************************************************************************}

{-----------------------------------------------------------------}
Procedure AjouterId (NomId : alpha; ObjId : Objet; TypId : Typage);
{-----------------------------------------------------------------}
{<<<
    Ajouter une variable simple dans la table des informations
    (en fin de table)
>>>}
Begin

   FinIdTab := succ (FinIdTab);
   With IdTab [FinIdTab] do
      begin
         Nom      := NomId;
         Obj      := ObjId;
         Typ      := TypId;
         Styp     := simple;
         Existval := false;
      end;

End;  { AjouterId }


{-------------------------------------------------------------------------}
Procedure AjouterAr (NomTab : alpha; ObjId : Objet; TypId : Typage;
                                                    DebAr,FinAr : Byte);
{-------------------------------------------------------------------------}
{<<<
    Ajouter une variable tableau dans la table des informations
    (en fin de table)
>>>}
var
     I,
     K     : integer;
     provi : str2;

Begin

   FinIdTab := pred (FinIdTab);
   K := 1;
   while (NomTab [K] <> ' ') and (K <= 8) do
   K := K + 1;
   NomTab [K]     := '[';
   NomTab [K + 3] := ']';
   for I := DebAr to FinAr do
```

```
      begin
         str (I:2, provi);
         NomTab [K + 1] := provi [1];
         NomTab [K + 2] := provi [2];
         FinIdTab := succ (FinIdTab);
         With IdTab [FinIdTab] do
         begin
            Nom      := NomTab;
            Obj      := ObjId;
            Typ      := TypId;
            Styp     := arrayty;
            Existval := false;
         end;
      end;

End;


{-------------------------------------------------}
Procedure MettreTyp (NumId : byte; TypId : Typage);
{-------------------------------------------------}
{<<<
    definit le type d'une variable
>>>}
Begin

   With IdTab [NumId] do
      begin
         Typ      := TypId;
      end;

End;  { MettreTyp }


{-------------------------------------}
Function ExistId (NomId : alpha) : byte;
{-------------------------------------}
{<<<
    retourne le numero de la variable si la variable existe
    sinon retourne 0
>>>}
var
     I   : integer;

Begin

   for I := 1 to FinIdTab do
      if NomId = IdTab [I] . Nom then
         begin
            ExistId := I;
            exit;
         end;
   ExistId := 0;

End;
```

```
{-------------------------------------------------}
Function ModifierVal (NomId : alpha; var Val) : byte;
{-------------------------------------------------}
{<<<
    modifie la valeur d'une variable qui existe
>>>}
var
     NumId : byte;
     VEnt  : integer absolute Val;
     VBool : boolean absolute Val;

Begin

   NumId := ExistId ( NomId);
   if NumId <> 0 then
      with IdTab [NumId] do
         begin
            if ExistVal = false then
               ExistVal := true;
            Case Typ of
               intty   : valint  := VEnt;
               boolty  :valbool := VBool;
            end; {case}
         end;
    ModifierVal := NumId;

End;


{-----------------------------------}
Function IdVal (NomId : alpha) : alpha;
{-----------------------------------}
{<<<
    retourne la valeur d'une variable qui existe,
    si la variable n'a pas de valeur definie alors
    retourne '????????????'
>>>}
var
     NumId  : byte;
     Chaine : alpha;

Begin

   NumId := ExistId ( NomId);
   if NumId <> 0 then
      with IdTab [NumId] do
         if ExistVal = false then
            IdVal := '????????????'
         else
            case Typ of
               intty  : begin
                           str(valint, chaine);
                           IdVal := chaine;
                        end;
               boolty : begin
                          case valbool of
                             false : IdVal := 'FALSE';
                             true  : IdVal := 'TRUE';
```

```
                        end; {case}
                      end;
            end  {case}
   else
      NoErreur (3);

End;
```

A.4.6 VISUAMC.INC.

```
{ Listage de VISUAMC.INC }
{*****************************************************************************}
{*                                                                           *}
{*               Primitives pour PROGRAIS relatives aux                      *}
{*                                                                           *}
{*                              M O D U L E                                  *}
{*                                                                           *}
{*          E D I T E U R  D E  M E M O I R E  C E N T R A L E                *}
{*                                                                           *}
{*****************************************************************************}

{--------------------- MEMOIRE  CENTRALE  OPERATIONS ---------------------}
{--------------------------------------}
Function AfficherOp (Debut : byte) : byte;
{--------------------------------------}
{<<<
    affiche une page d'operations a partir de l'element
    No. Debut de la table des operations et
    retourne la valeur du debut d'affichage courant
>>>}
var
     I,
     J        : byte;

Begin
   if NumFenCour <> 2 then
      FenCharge(2);
   Attribut := CoulOp;
   I := Debut;
   for J := (OpYH  + 1) to (OpYH + 13) do
      begin
         GoXY(OpXG,J);
         EcrisChn(CodeBlc);
         If I <= FinOpTab then
            begin
               GoXY(OpXG,J);
               EcrisChn(OpTab[I].Code);
            end;
         I := succ(I);
      end;
   AfficherOp := Debut;

End;  { AfficherOp }
```

```
{---------------------------------------}
Function NouvelleLigne(Num : byte) : byte;
{---------------------------------------}
{<<<
    remet la ligne precedente en video normale,
    place la nouvelle ligne en survideo avec changement de page
        d'affichage si la nouvelle ligne n'est pas affichee,
    retourne Num.
>>>}
var
   Y        : integer;
   LignCour : LigneCode;

Begin

   if NumFenCour <> 2 then
      FenCharge (2);
   if NoLigne = 0 then
      begin
         Attribut := InvCoulOp;
         GoXY (OpXG, OpYH + 1);
         EcrisChn (CodeBlc);
         GoXY (OpXG, OpYH + 1);
         EcrisChn (OpTab[1].Code);
         Attribut := CoulOp;
         NouvelleLigne := 1;
      end
   else
      if Num <> NoLigne then
         begin
            Y := NoLigne - DebAffOp + 1;
            Attribut := CoulOp;
            GoXY (OpXG, OpYH + Y);
            EcrisChn (CodeBlc);
            GoXY (OpXG, OpYH + Y);
            EcrisChn (OpTab[NoLigne].Code);
            if not (Num in [DebAffOp..(DebAffOp + 12)]) then
               begin
                  case Num of
                   1 : begin
                          Debaffop := Num;
                          Y := 1;
                       end;
                   2 : begin
                          DebAffOp := Num - 1;
                          Y := 2;
                       end;
                   else
                       begin
                          Debaffop := Num - 2;
                          Y:= 3;
                       end;
                  end;
                  Attribut := CoulOp;
                  DebAffOp := AfficherOp (DebAffOp);
               end
            else
               Y := Num - DebAffOp + 1;
```

```
            Attribut := InvCoulOp;
            GoXY (OpXG, OpYH + Y);
            EcrisChn (CodeBlc);
            GoXY (OpXG, OpYH + Y);
            EcrisChn (OpTab[Num].Code);
            Attribut := CoulOp;
            NouvelleLigne := Num;
         end;

End;  { NouvelleLigne }


{-------------------- MEMOIRE CENTRALE INFORMATIONS --------------------}
{-----------------------------------------------}
Procedure ChangerCase (NumId, NumCase, Cl : byte);
{-----------------------------------------------}
{<<<
    affiche dans la case NumCase les Nom, type et valeur
    de l'element NumId de la table des identificateurs
          avec clignotement si Cl = 1,
          sans clignotement si Cl = 0,
         (couleur suivant la nature du type)
>>>}
var
     I      : byte;
     Chaine : Alpha;

Begin

   with IdTab [NumId] do
      for I := Cl downto 0 do
         begin
            if STyp = simple then
               Attribut := CoulId + (128 * I)
            else
               Attribut := CoulAr + (128 * I);
            GoXY (IdXNom [NumCase], IdYNom [NumCase]);
            EcrisChn (Nom);
            GoXY (IdXAr [NumCase], IdYAr [NumCase]);
            EcrisChn (ArBlc);
            GoXY (IdXTy  [NumCase], IdYTy  [NumCase]);
            Case Typ of
              pasty  : Chaine := TyBlc;
              intty  : Chaine := 'Entier ';
              boolty : Chaine := 'Booléen';
            end; { case }
            EcrisChn (Chaine);
            GoXY (IdXVal [NumCase], IdYVal [NumCase]);
            if ExistVal then
               case Typ of
                 pasty  : Chaine := '     --     ';
                 intty  : str (valint:12, Chaine);
                 boolty : begin
                             if valbool then
                                Chaine := '        VRAI'
                             else
                                Chaine := '        FAUX';
                          end;
```

```
            end  { case }
         else
            Chaine := PasVal;
         EcrisChn (Chaine);
         if I = 1 then delay (1500);
      end;  { for }

End;


{---------------------------------------}
Procedure MettreCaseVide (NumCase : byte);
{---------------------------------------}
{<<<
    affiche dans la case '??' indiquant la fin de la table
    des identificateurs.
>>>}
var
     I      : byte;
     Chaine : Alpha;

Begin

   Attribut := CoulVide;
   GoXY (IdXNom [NumCase], IdYNom [NumCase]);
   EcrisChn (NomBlc);
   GoXY (IdXAr [NumCase], IdYAr [NumCase]);
   EcrisChn (ArBlc);
   GoXY (IdXTy  [NumCase], IdYTy  [NumCase]);
   EcrisChn (CaseVide);
   GoXY (IdXVal [NumCase], IdYVal [NumCase]);
   EcrisChn (NomBlc);

End;


{-----------------------------------------------------}
Procedure ChangerPage (NumId : byte; NumLigne : integer);
{-----------------------------------------------------}
{<<<
    affiche une nouvelle page de la table des identificateurs.
>>>}

var
     I   : integer;
     Num : integer;
     XX  : integer;

Begin

   I := NumLigne - DebAffId;
   Case I of
     2    : DebAffId := DebAffId + 1;
     3    : DebAffId := DebAffId + 2;
     else
            DebAffId := NumLigne;
   end; { case }
   Num := (DebAffId * 3) - 2;
```

```
   XX  := 0;
   for I := 1 to 6 do
      begin
         if Num <= finIdTab then
            begin
               ChangerCase(Num, I, 0);
               if Num = NumId then
                  XX := I;
            end
         else
            MettreCaseVide (I);
         Num := succ (Num);
      end;  { for }
   if XX <> 0 then
      ChangerCase(NumId, XX, 1);   { fait clignoter la case de NumId }

End;


{--------------------------------}
Procedure AfficherId (Num : byte);
{--------------------------------}
{<<<
    met a jour l'affichage du Nom, du type et de la valeur
        de l'element num de la table des identificateurs
        avec changement de la page d'affichage si l'element
        concerne n'est pas dans la page courante,
    en cas de nouvelle page, la variable DebAffId est
        modifiee en consequence.
>>>}
var
     NumCase  : byte;
     I        : integer;

Begin

   if NumFenCour <> 2 then
      FenCharge(2);
   if DebAffId = 0 then
      ChangerPage (Num, 1)
   else
      begin
         I := (Num + 2) div 3;    { No. ligne dans la table virtuelle }
         if I = DebAffId then
            begin
               NumCase := (Num - 1) mod 3 + 1;
               ChangerCase (Num, NumCase, 1);
            end
         else
            if I = DebAffId + 1 then
               begin
                  NumCase := (Num - 1) mod 3 + 4;
                  ChangerCase (Num, NumCase, 1);
               end
            else
               ChangerPage (Num, I);
      end;
```

```
End;  { AfficherId }
```

A.4.7 SCANNER.INC.

```
{Listage de scanner.inc}
{*****************************************************************************}
{*                                                                           *}
{*               Primitives pour PROGRAIS relatives aux                      *}
{*                                                                           *}
{*                            M O D U L E                                    *}
{*                                                                           *}
{*                                                                           *}
{*                 A N A L Y S E U R   L E X I C A L                         *}
{*                                                                           *}
{*****************************************************************************}


{-----------------------------------------------------}
Function SymScan (var NoLigne, Nopos : integer) : Symbole;
{-----------------------------------------------------}
{<<<
    fait l'analyse lexicale du symbole suivant dans le texte

    Arguments : NoLigne, LigneCour et NoPos.
    Resultats : Symscan,
           si SymScan = ident alors identificateur dans Id  ,
           si SymScan dans KwSym alors mot reserve dans Id  ,
           si SymScan = entcon alors valeur entiere dans entss
                  ainsi que chaine corresp. dans Id,
                  actualisation de NoLigne, LigneCour et NoPos.
    Remarque  : si Symscan = modif, il faut passer a la ligne suivante.
>>>}
Var
     I,
     J,
     K    : integer;
     Ch   : char;

Begin

   { lit le caractere a la position NoPos de la ligne NoLigne : }
   Ch := LigneCour [NoPos];

   { traitement des caracteres blancs : }
   While Ch = ' ' do
      begin
         NoPos := succ (NoPos);
         Ch := LigneCour [NoPos];
      end;

   Case Ch of
   { traitement identificateur et mot reserve : }
   'a'..'z','_','A'..'Z':     { Lettre }
      begin
         K := 0;
```

```
      Id := '';
      repeat
         if K < LgId then
            begin
               K := K+1;
               Id := Id + Ch;
            end;
         Nopos := Succ (NoPos);
         Ch := LigneCour [NoPos];
      until not (Ch in ['0'..'9','a'..'z','_','A'..'Z']);
                                                {LettreOuChiffre}
      Id := Upper (Id);    { conversion en majuscules }
      while length (Id) < LgId do
         Id := Id + ' ';   { ajout de blancs }

    { recherche binaire pour mot reserve : }
      I := 1;
      J := NbrKw;
      repeat
         K := (I + J) div 2;
         if Id <= Kw [K] then
            J := K - 1;
         if Id >= Kw [K] then
            I := K + 1;
      until I > J;

    { resultat de SymScan : }
      if I - 1 > J then
         SymScan := KwSym [K]   { mot reserve }
      else
         SymScan := ident;      { identificateur }

    { alternative avec recherche sequentielle :
      J := 0;
      for I := 1 to NbrKw do
          if Id = Kw [I] then
             J := I;
      if J = 0 then
         SymScan := ident
      else
         SymScan := KwSym [J]; }

   end;

{ traitement nombre entier : }
'0'..'9':    { chiffre }
   begin
      entss := 0;
      SymScan := entcon;    { nombre entier }
      repeat
         entss := entss * 10 + ord (Ch) - ord ('0');
         NoPos := succ (NoPos);
         Ch := LigneCour [NoPos];
      until (Ch < '0') or (Ch > '9');
      str (entss : 12 , Id);
   end;

{ traitement des symboles doubles : }
```

```
':' : begin
         NoPos := succ (NoPos);
         Ch := LigneCour [NoPos];
         if Ch = '=' then
            begin
               SymScan := affect;  { := }
               NoPos := succ (NoPos);;
            end
         else
            SymScan := deuxpt;   { : }
         end;
'<' : begin
         NoPos := succ (NoPos);
         Ch := LigneCour [NoPos];
         if Ch = '>' then
            begin
               SymScan := nonegal;  { <> }
               NoPos := succ (NoPos);
            end
         else
            if Ch = '=' then
               begin
                  SymScan := ppetiteg;  { <= }
                  NoPos := succ (NoPos);
               end
            else
               SymScan := ppetit;  { < }
         end;
'>' : begin
         NoPos := succ (NoPos);
         Ch := LigneCour [NoPos];
         if Ch = '=' then
            begin
               SymScan := pgrandeg;  { >= }
               Nopos := succ (NoPos);
            end
         else
            SymScan := pgrand;  { > }
         end;
'.' : begin
         NoPos := succ (NoPos);
         Ch := LigneCour [NoPos];
         if Ch = '.' then
            begin
               SymScan := ptpt; { .. }
               NoPos := succ (NoPos);
            end
         else
            SymScan := point;  { . }
         end;
'(' , ')' , '*' , '+' , '-' , ',' , ';' , '[' , ']':
      begin
         SymScan := Ssp [Ch];
         NoPos := Succ (NoPos);
      end;
#13 : Symscan := finlig;
else
      NoErreur (3);
```

```
   end; { end case }

End;   { SymScan }
```

A.4.8 ANALOP.INC.

```
{ Listage de ANALOP.INC }
{*************************************************************************}
{*                                                                       *}
{*               Primitives pour PROGRAIS relatives aux                  *}
{*                                                                       *}
{*                            M O D U L E                                *}
{*                                                                       *}
{*               A N A L Y S E U R   S Y N T A X I Q U E                 *}
{*                                                                       *}
{*************************************************************************}


{-----------------------------------------}
function TestFinLigne (Sym: Symbole): boolean;
{-----------------------------------------}
{<<<
    retourne "vrai" si le symbole est un symbole de fin de ligne
    sinon retorne "faux"
>>>}
var
     num : integer;

Begin

   if sym = finlig then
      begin
         num          := succ (NoLigne);
         NoLigne      := NouvelleLigne (num);
         NoPos        := 1;
         LigneCour    := OpTab [NoLigne] . Code + #13;
         TestFinLigne := true;
      end
   else
      TestFinLigne := false;

End;


{-----------------------------------}
Procedure AcceptScan (SymOk : Symbole);
{-----------------------------------}
{<<<
    lit le symbole suivant
    provoque une sortie d'erreur si le symbole n'est pas compatible
       avec le langage DEMO PASCAL
>>>}
Begin

   if Sym = SymOk then
```

```
      begin
         Biiip;
         Sym := SymScan (NoLigne,NoPos);
         while TestFinLigne (Sym) do
            begin
               Sym := SymScan (NoLigne,NoPos);
            end;
      end
   else
      NoErreur (3);  {pas compatible avec DEMO PASCAL}

End;


{------------------}
Procedure AcceptFin;     { point final }
{------------------}

Begin

   if sym <> point then
      NoErreur (3);

End;

{<<<<<<<<<<<<<
       Les procedures qui suivent se rapportent aux regles
       de la syntaxe concrete EBNF du langage DEMO PASCAL
>>>>>>>>>>>>>>}

{--------------------}
Procedure Expression;          { regle 27 }
{--------------------}

Begin

   case sym of
     entcon : AcceptScan (entcon);
     ident  : AcceptScan (ident);
     else
              NoErreur (3);
   end; { case }

End;


{------------------}
Procedure Condition;          {règle 26}
{------------------}

Const
     Relop : Symset = [egal, nonegal, ppetit, ppetiteg, pgrand, pgrandeg];

Begin

   Expression;
   if Erreur then exit;
   if sym in RelOp then
```

```
      AcceptScan (sym)
   else
      NoErreur (3);
   if Erreur then exit;
   Expression;

End;


{-------------------}
Procedure IdVariable;                    { regle 25 }
{-------------------}

Begin

   AcceptScan (ident);
   if Erreur then exit;
   if Sym = gcroch then
      begin
         AcceptScan (gcroch);
         Expression;
         if Erreur then exit;
         AcceptScan (dcroch);
      end;

End;


{------------------------}
Procedure ListeExpression;               { regle 24 }
{------------------------}

Begin

   Expression;
   if Erreur then exit;
   while Sym = virg do
      begin
         AcceptScan (virg);
         Expression;
         if Erreur then exit;
      end;

End;


{---------------------}
Procedure ListeVariable;                 { regle 23 }
{---------------------}

Begin

   IdVariable;
   if Erreur then exit;
   while Sym = virg do
      begin
         AcceptScan (virg);
         IdVariable;
```

```
         if Erreur then exit;
      end;

End;


{--------------------------}
Procedure Operation; forward;
{--------------------------}


{---------------------------}
Procedure OperationRepetitive;          { regle 22 }
{---------------------------}

Begin

   AcceptScan (whilekw);
   if Erreur then exit;
   Condition;
   if Erreur then exit;
   AcceptScan (dokw);
   if Erreur then exit;
   Operation;

End;

{-------------------------------}
Procedure OperationConditionnelle;      { regle 21 }
{-------------------------------}

Begin

   AcceptScan (ifkw);
   Condition;
   if Erreur then exit;
   AcceptScan (thenkw);
   if Erreur then exit;
   Operation;
   if Erreur then exit;
   AcceptScan (elsekw);
   Operation;

End;


{-------------------------}
Procedure OperationEcriture;            { regle 20 }
{-------------------------}

Begin

   writeln ('      operation d''écriture');
   AcceptScan (writekw);
   AcceptScan (gparen);
   if Erreur then exit;
   ListeExpression;
   if Erreur then Exit;
```

```
   AcceptScan (dparen);

End;


{-------------------------}
Procedure OperationLecture;            { regle 19 }
{-------------------------}

Begin

   writeln ('      opération de lecture');
   AcceptScan (readkw);
   AcceptScan (gparen);
   if Erreur then exit;
   ListeVariable;
   if Erreur then Exit;
   AcceptScan (dparen);

End;


{----------------------------}
Procedure OperationAffectation;        { regle 18 }
{----------------------------}

Begin

   writeln ('      opération d''affectation');
   IdVariable;
   if Erreur then exit;
   AcceptScan (affect);
   if erreur then exit;
   Expression;

End;


{----------------------}
Procedure OperationVide;               { regle 17 }
{----------------------}

Begin

End;


{----------------------------------}
Procedure OperationComposee; forward;
{----------------------------------}


{------------------}
Procedure Operation;                   { regles 14, 15 et 16 }
{------------------}

Begin
```

```
   Case Sym of
     endkw   : OperationVide;
     ptvirg  : OperationVide;
     ident   : OperationAffectation;
     readkw  : OperationLecture;
     writekw : OperationEcriture;
     beginkw : OperationComposee;
     ifkw    : OperationConditionnelle;
     whilekw : OperationRepetitive;
     else
        NoErreur (3);
   end; { case }

End;


{----------------------}
Procedure ListeOperation;              { regle 13 }
{----------------------}

Begin

   Operation;
   while Sym = ptvirg do
      begin
         AcceptScan (ptvirg);
         Operation;
      end;

End;



{-------------------------}
Procedure OperationComposee;           { regle 12 }
{-------------------------}

Begin

   writeln ('    opération composée');
   AcceptScan (beginkw);
   if Erreur then Exit;
   ListeOperation;
   if Erreur then exit;
   AcceptScan (endkw);

End;


{------------------------}
Procedure PartieOperations;            { regle 11 }
{------------------------}

Begin

   OperationComposee;

End;
```

```
{------------------}
Procedure TypeSimple;                    { regle 10 }
{------------------}

Begin

   if Sym = integerkw then
      begin
         AcceptScan (integerkw);
         writeln ('    de type entier');
      end
   else
      begin
         AcceptScan (booleankw);
         writeln ('    de type bouléen');
      end;

End;


{--------------------}
Procedure TypeTableau;                   { regle 9  }
{--------------------}

Begin

   AcceptScan (arraykw);
   writeln ('    de type tableau');
   AcceptScan (gcroch);
   if Erreur then exit;
   AcceptScan (entcon);
   if Erreur then exit;
   AcceptScan (ptpt);
   if Erreur then exit;
   AcceptScan (entcon);
   if Erreur then exit;
   AcceptScan (dcroch);
   if Erreur then exit;
   AcceptScan (ofkw);
   if Erreur then exit;
   TypeSimple;

End;


{-----------------}
Procedure DeclType;                      { regle 8 }
{-----------------}

Begin

   if Sym = arraykw then
      TypeTableau
   else
      TypeSimple;
```

```
End;


{----------------}
Procedure ListeId;                    { regle 7 }
{----------------}

Begin

   AcceptScan (ident);
   writeln ('    ',id);
   if Erreur then Exit;
   while Sym = virg do
      begin
         AcceptScan (virg);
         AcceptScan (ident);
         writeln ('    ',id);
         if Erreur then exit;
      end;

End;


{----------------------}
Procedure DeclVariables;              { regle 6 }
{----------------------}

Begin

   ListeId;
   If Erreur then exit;
   AcceptScan (deuxpt);
   if Erreur then Exit;
   DeclType;
   if Erreur then Exit;
   AcceptScan (Ptvirg);

End;


{--------------------------}
Procedure ListeDeclVariables;         { regle 5 }
{--------------------------}

Begin

   repeat
      DeclVariables;
      if Erreur then exit;
   until Sym <> ident;

End;


{--------------------------}
Procedure PartieDeclarations;         { regle 4 }
{--------------------------}
```

```
Begin

   if Sym = varkw then
      begin
         AcceptScan (varkw);
         ListeDeclVariables;
      end;

End;


{-------------}
Procedure Bloc;                          { regle 3 }
{-------------}

Begin

   writeln ('  partie : déclarations');
   PartieDeclarations;
   if Erreur then exit;
   writeln ('  partie : opérations');
   PartieOperations;

End;


{---------------}
Procedure EnTete;                        { regle 2 }
{---------------}

Begin

   AcceptScan (ProgramKw);
   if Erreur then exit;
   AcceptScan (Ident);
   if Erreur then exit;
   NomProg := Id;
   AcceptScan (Ptvirg);

end;


{------------------}
Procedure Programme;                     { regle 1 }
{------------------}

Begin

   writeln ('en-tete du programme');
   EnTete;
   if Erreur then Exit;
   writeln ('bloc principal');
   Bloc;
   if Erreur then Exit;
   writeln ('fin du programme');
   AcceptFin;

End;  { Programme }
```

```
{-----------------------}
Procedure AmorcerAnalSynt;
{-----------------------}

Begin

   writeln ('ANALYSE de la SYNTAXE du programme :');
   writeln ('---------------------------------');
   NoLigne := NouvelleLigne(1);
   NoPos := 1;
   LigneCour := OpTab [NoLigne] . Code + #13;
   Biiip;
   sym := SymScan (NoLigne, NoPos);
   while TestFinLigne (Sym) do
      begin
         Biiip;
         Sym := SymScan (NoLigne,NoPos);
      end;

End;   { AmorcerAnalSynt }


{----------------------}
procedure TestAnalyseur;
{----------------------}

var
     Ch  : char;

Begin

   AmorcerAnalSynt;
   Programme;
   read (kbd,Ch);

End;  { TestAnalyseur }
```

A.4.9 EXECPROG.INC.

```
{ Listage de EXECPROG.INC }
{*************************************************************************}
{*                                                                       *}
{*               Primitives pour PROGRAIS relatives aux                  *}
{*                                                                       *}
{*                          M O D U L E S                                *}
{*                                                                       *}
{*              A N A L Y S E U R   S Y N T A X I Q U E                  *}
{*                          et                                           *}
{*             E X E C U T E U R  de  P R O G R A M M E                  *}
{*                                                                       *}
{*************************************************************************}


{----------------------}
```

```
Procedure InspecterIdTab;
{----------------------}
{<<<
     Fait a la demande l'inspection de la
     partie information de la Memoire Centrale
>>>}
var
   TC       : Char;
   NumLigne : integer;

Begin
   NoOpt (2);
   if NumFenCour <> 2 then
      Fencharge (2);
   Repeat
      TC := AttTouche;
      if TC = FB { fleche bas } then
         begin
            NumLigne := DebaffId + 1;
            ChangerPage (0, NumLigne);
         end;
      if TC = FH { fleche haut } then
         begin
            if (DebAffId = 0) or (DebAffId = 1) then
               Bip
            else
               begin
                  NumLigne := DebAffId - 1;
                  ChangerPage (0, NumLigne);
               end;
         end;
   until TC = Esc;
   NoOpt (1);
End;  { InspecterIdTab }


{-----------------------------------------}
function TestFinLigne (Sym: Symbole): boolean;
{-----------------------------------------}
{<<<
    retourne "vrai" si le symbole est un symbole de fin de ligne
    sinon retorne "faux"
>>>}

var
     num : integer;

Begin

   if sym = finlig then
      begin
         num          := succ (NoLigne);
         NoLigne      := NouvelleLigne (num);
         NoPos        := 1;
         LigneCour    := OpTab [NoLigne] . Code + CR;
         TestFinLigne := TRUE;
      end
   else
```

```
      TestFinLigne := FALSE;

End;   { TestFinLigne }


{-------------------------------------}
Procedure AcceptScan (SymOk : Symbole);
{-------------------------------------}
{<<<
    lit le symbole suivant
    provoque une sortie d'erreur si le symbole n'est pas compatible
       avec le langage DEMO PASCAL
>>>}
Begin

   if Sym = SymOk then
      begin
         Sym := SymScan (NoLigne,NoPos);
         while TestFinLigne (Sym) do
            begin
               Sym := SymScan (NoLigne,NoPos);
            end;
      end
   else
      NoErreur (3);
   if Touche = Esc then
      InspecterIdTab;

End;   { AcceptScan }


{-----------------}
Procedure AcceptFin;     { point final }
{-----------------}

Begin

   if sym <> point then
      NoErreur (3);

End;


{--------------------}
Procedure AmorcerExec;
{--------------------}

Begin

   NoLigne := NouvelleLigne(1);
   NoPos := 1;
   LigneCour := OpTab [NoLigne] . Code + CR;
   sym := SymScan (NoLigne, NoPos);
   while TestFinLigne (Sym) do
      begin
         Sym := SymScan (NoLigne,NoPos);
      end;
```

```
End;   { AmorcerExec }


{<<<<<<<<<<<<<
       Les procedures qui suivent se rapportent aux regles
       de la syntaxe concrete EBNF du langage DEMO PASCAL
>>>>>>>>>>>>>>}

{-------------------}
Procedure Expression;                  { regle 27 }
{-------------------}
var
     code  : integer;

Begin

   case sym of
     entcon : begin
                 AcceptScan (entcon);
                {!}
                 ValExpr := entss;
                 str (entss, ChnExpr);
                {!}
              end;
     ident  : begin
                 AcceptScan (ident);
                {!}
                 ChnExpr := IdVal (Id);
                 val (ChnExpr, ValExpr, code);
                {!}
              end;
     else
              NoErreur (3);
   end; { case }

End;   { Expression }


{------------------}
Procedure Condition;                   { regle 26 }
{------------------}

Const
     Relop : Symset = [egal, nonegal, ppetit, ppetiteg, pgrand, pgrandeg];

Begin

   Expression;
   if Erreur then exit;
   if sym in RelOp then
      AcceptScan (sym)
   else
      NoErreur (3);
   if Erreur then exit;
   Expression;

End;
```

```
{------------------}
Procedure IdVariable;                    { regle 25 }
{------------------}

Begin

   AcceptScan (ident);
  {!}
   NomIdent := Id;
  {!}
   if Erreur then exit;
   if Sym = gcroch then
      begin
         AcceptScan (gcroch);
         Expression;
         if Erreur then exit;
         AcceptScan (dcroch);
      end;

End;


{-----------------------}
Procedure ListeExpression;               { regle 24 }
{-----------------------}

Begin

   Expression;
   if Erreur then exit;
   while Sym = virg do
      begin
        {!}
         NoErreur (3);
        {!}
         AcceptScan (virg);
         Expression;
         if Erreur then exit;
      end;

End;


{---------------------}
Procedure ListeVariable;                 { regle 23 }
{---------------------}

Begin

   IdVariable;
   if Erreur then exit;
   while Sym = virg do
      begin
         AcceptScan (virg);
         IdVariable;
         if Erreur then exit;
      end;
```

```
End;


{-------------------------}
Procedure Operation; forward;
{-------------------------}


{---------------------------}
Procedure OperationRepetitive;          { regle 22 }
{---------------------------}

Begin

   AcceptScan (whilekw);
   if Erreur then exit;
   Condition;
   if Erreur then exit;
   AcceptScan (dokw);
   if Erreur then exit;
   Operation;

End;

{-------------------------------}
Procedure OperationConditionnelle;      { regle 21 }
{-------------------------------}

Begin

   AcceptScan (ifkw);
   Condition;
   if Erreur then exit;
   AcceptScan (thenkw);
   if Erreur then exit;
   Operation;
   if Erreur then exit;
   AcceptScan (elsekw);
   Operation;

End;


{-------------------------}
Procedure OperationEcriture;            { regle 20 }
{-------------------------}

var
     Chaine  : alpha;

Begin

  {!}
   NoComm (7);
   delay (1000);
  {!}
   AcceptScan (writekw);
```

```
   AcceptScan (gparen);
   if Erreur then exit;
   ListeExpression;
   if Erreur then Exit;
   AcceptScan (dparen);
  {!}
   FenCharge (1);
   write (ChnExpr);
   EcrisChn (ChnExpr);
   delay (3000);
   Fencharge (5);
   Fencharge (6);
   Fencharge (2);
   delay (3000);
  {!}

End;


{------------------------}
Procedure OperationLecture;            { regle 19 }
{------------------------}
var
     code  : integer;

Begin

  {!}
   NoComm (6);
   delay (1000);
  {!}
   AcceptScan (readkw);
   AcceptScan (gparen);
   if Erreur then exit;
   ListeVariable;
   if Erreur then Exit;
   AcceptScan (dparen);
  {!}
   FenCharge (1);
   read (ChnExpr);
   EcrisChn (ChnExpr);
   val (ChnExpr, ValExpr, code);
   FenCharge (5);
   Fencharge (6);
   if code = 0 then
      AfficherId ( ModifierVal (NomIdent,ValExpr))
   else
      NoErreur (3);
  {!}

End;


{---------------------------}
Procedure OperationAffectation;        { regle 18 }
{---------------------------}

Begin
```

```
  {!}
   NoComm (5);
   delay (1000);
  {!}
   IdVariable;
   if Erreur then exit;
   AcceptScan (affect);
   if erreur then exit;
   Expression;
  {!}
   AfficherId ( ModifierVal (NomIdent,ValExpr));
  {!}

End;


{---------------------}
Procedure OperationVide;                { regle 17 }
{---------------------}

Begin

End;


{---------------------------------}
Procedure OperationComposee; forward;
{---------------------------------}


{------------------}
Procedure Operation;                    { regles 14, 15 et 16 }
{------------------}

Begin

   Case Sym of
     endkw   : OperationVide;
     ptvirg  : OperationVide;
     ident   : OperationAffectation;
     readkw  : OperationLecture;
     writekw : OperationEcriture;
     beginkw : OperationComposee;
     ifkw    : OperationConditionnelle;
     whilekw : OperationRepetitive;
     else
        NoErreur (3);
   end; { case }

End;


{----------------------}
Procedure ListeOperation;               { regle 13 }
{----------------------}

Begin
```

```
   Operation;
   while Sym = ptvirg do
      begin
         AcceptScan (ptvirg);
         Operation;
      end;

End;


{-------------------------}
Procedure OperationComposee;          { regle 12 }
{-------------------------}

Begin

   AcceptScan (beginkw);
   if Erreur then Exit;
   ListeOperation;
   if Erreur then exit;
   AcceptScan (endkw);

End;


{-------------------------}
Procedure PartieOperations;           { regle 11 }
{-------------------------}

Begin

  {!}
   Curseur (true);
  {!}
   OperationComposee;

End;


{-------------------}
Procedure TypeSimple;                 { regle 10 }
{-------------------}

var
     I : Byte;

Begin

   if Sym = integerkw then
      begin
         AcceptScan (integerkw);
        {!}
         for I := DebDecl to FinIdTab do
            begin
               MettreTyp (I, intty);
               AfficherId (I);
```

```
            end;
         {!}
      end
   else
      begin
         AcceptScan (booleankw);
        {!}
         for I := DebDecl to FinIdTab do
            begin
               MettreTyp (I, boolty);
               AfficherId (I);
            end;
        {!}
      end;

End;


{--------------------}
Procedure TypeTableau;                  { regle 9  }
{--------------------}

var
     I,
     DebAr,
     FinAr  : integer;
     NomTab : alpha;

Begin
  {!}
   if DebDecl <> FinIdtab then
      begin
         NoErreur (3);
         exit;
      end;
  {!}
   AcceptScan (arraykw);
   AcceptScan (gcroch);
   if Erreur then exit;
   AcceptScan (entcon);
   if Erreur then exit;
  {!}
   DebAr := Entss;
  {!}
   AcceptScan (ptpt);
   if Erreur then exit;
   AcceptScan (entcon);
   if Erreur then exit;
  {!}
   FinAr := Entss;
   if (FinAr < DebAr) or (FinAr > 99) then
      begin
         NoErreur (3);
         exit;
      end;
   NomTab := IdTab [FinIdTab] . Nom;
   AjouterAr (NomTab, variable, PasTy, DebAr, FinAr);
{  for I := DebDecl to FinIdTab do
```

```
      AfficherId (I);  }
  {!}
   AcceptScan (dcroch);
   if Erreur then exit;
   AcceptScan (ofkw);
   if Erreur then exit;
   TypeSimple;

End;


{-----------------}
Procedure DeclType;                    { regle 8 }
{-----------------}

Begin

   if Sym = arraykw then
      begin
        {!}
         NoComm (4);
        {!}
         TypeTableau;
      end
   else
      begin
        {!}
         NoComm (3);
        {!}
         TypeSimple;
      end;

End;


{----------------}
Procedure ListeId;                     { regle 7 }
{----------------}

Begin

   AcceptScan (ident);
   if Erreur then Exit;
  {!}
   NoComm (2);
   AjouterId ( id, variable, pasty);
   AfficherId (FinIdTab);
   DebDecl := FinIdTab;
  {!}
   while Sym = virg do
      begin
         AcceptScan (virg);
         AcceptScan (ident);
         if Erreur then exit;
        {!}
         NoComm (2);
         AjouterId ( id, variable, pasty);
         AfficherId (FinIdTab);
```

```
        {!}
      end;

End;


{----------------------}
Procedure DeclVariables;                { regle 6 }
{----------------------}

Begin

   ListeId;
   If Erreur then exit;
   AcceptScan (deuxpt);
   if Erreur then Exit;
   DeclType;
   if Erreur then Exit;
   AcceptScan (Ptvirg);

End;


{--------------------------}
Procedure ListeDeclVariables;           { regle 5 }
{--------------------------}

Begin

   repeat
      DeclVariables;
      if Erreur then exit;
   until Sym <> ident;

End;


{-------------------------}
Procedure PartieDeclarations;           { regle 4 }
{-------------------------}

Begin

   if Sym = varkw then
      begin
         AcceptScan (varkw);
         ListeDeclVariables;
      end;

End;


{-------------}
Procedure Bloc;                         { regle 3 }
{-------------}

Begin
```

```
  {!}
   NoPart (2);
   Biiip;
  {!}
   PartieDeclarations;
   if Erreur then exit;
  {!}
   NoPart (3);
   Biiip;
  {!}
   PartieOperations;

End;


{---------------}
Procedure EnTete;                    { regle 2 }
{---------------}

Begin

   AcceptScan (ProgramKw);
   if Erreur then exit;
   AcceptScan (Ident);
   if Erreur then exit;
  {!}
   NomProg := Id;
  {!}
   AcceptScan (Ptvirg);

end;


{-----------------}
Procedure Programme;                 { regle 1 }
{-----------------}

var
   Ch       : Char;
   NumLigne : integer;

Begin

  {!}
   NoPart (1);
   AmorcerExec;
   Biiip;
  {!}
   EnTete;
   if Erreur then exit;
   Bloc;
   if Erreur then exit;
   AcceptFin;
   if Erreur then exit;
  {!}
   NoPart (4);
   Bip;
   repeat
```

```
      TT := AttTouche;
      if TT = #27 then
         InspecterIdTab;
   until TT <> #27;
   NoFin;
  {!}

End;  { Programme }


{------------------}
procedure LanceExec;
{------------------}

Begin

   Programme;

End;  { LanceExec }
```

Annexe 3

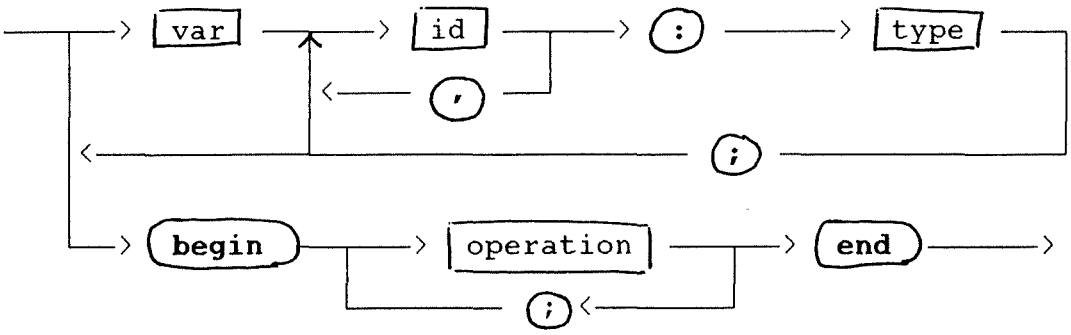
DIAGRAMMES SYNTAXIQUES DE LA SYNTAXE CONCRETE

Dans le chapitre 3, la syntaxe concrète du mini langage de programmation *DEMO PASCAL* de convention Pascal a été défini en utilisant le méta-langage de la forme EBNF(1). Dans cette annexe, nous donnons cette même syntaxe exprimée dans le méta-langage des diagrammes syntaxiques.

Programme

en_tete

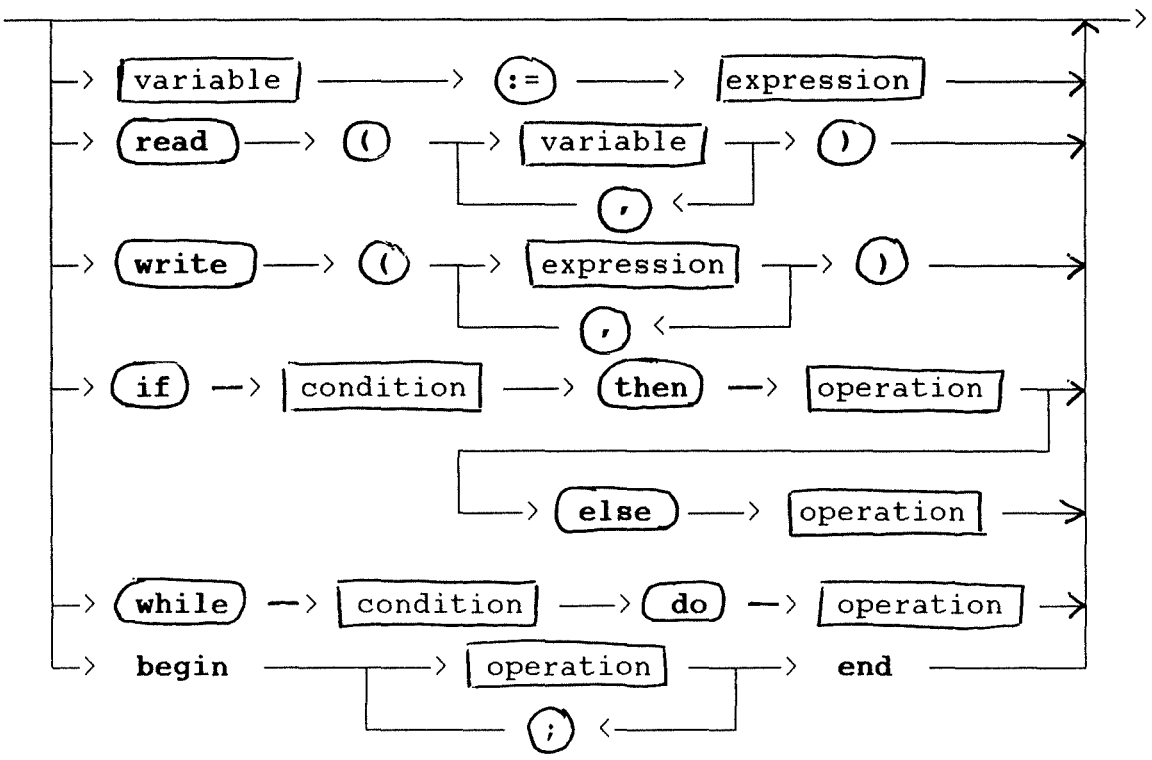
bloc



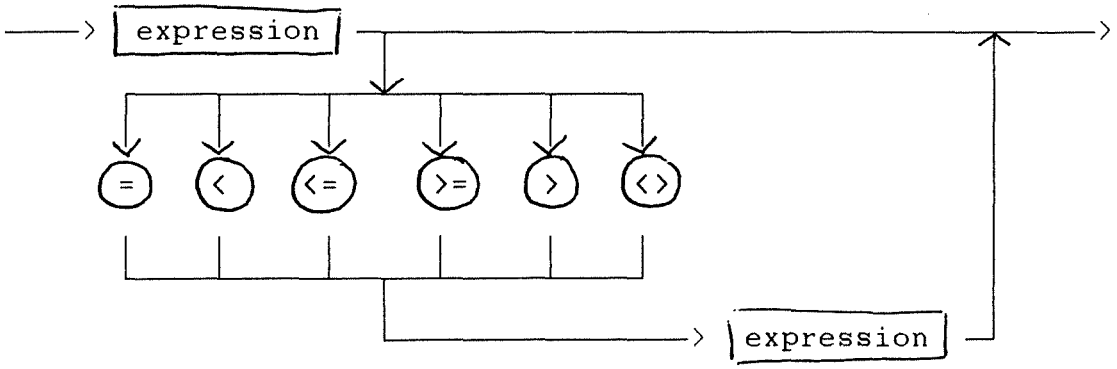
---

(1) Cf. *Table 3.3* du chapitre 3.

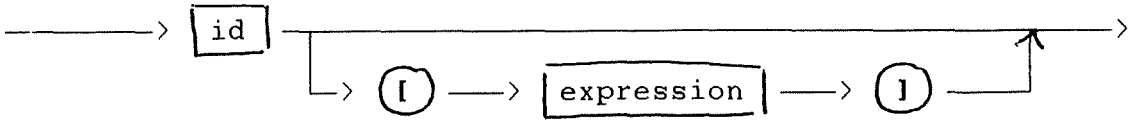
operation



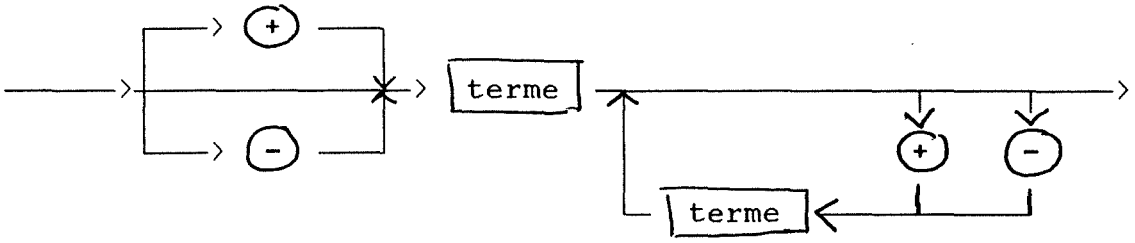
condition



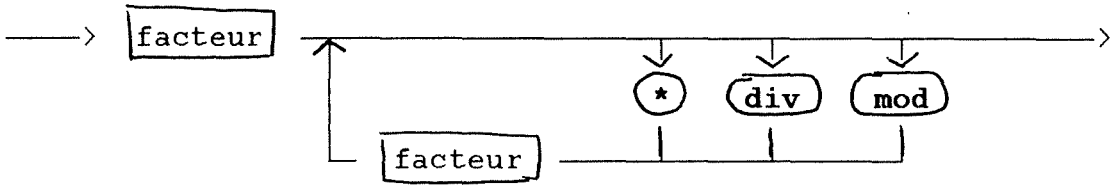
variable



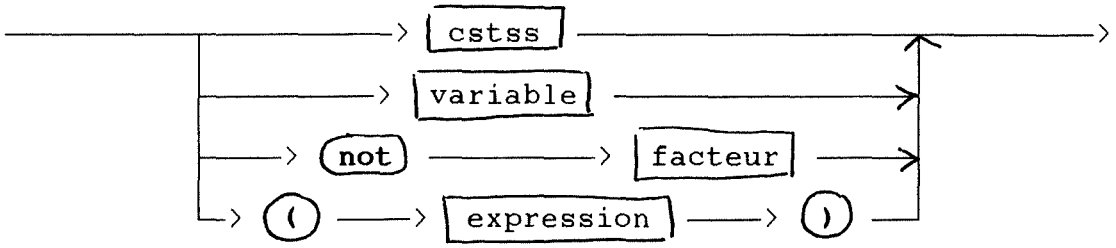
expression



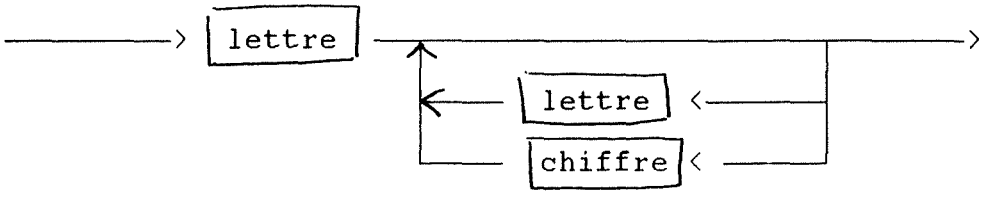
terme



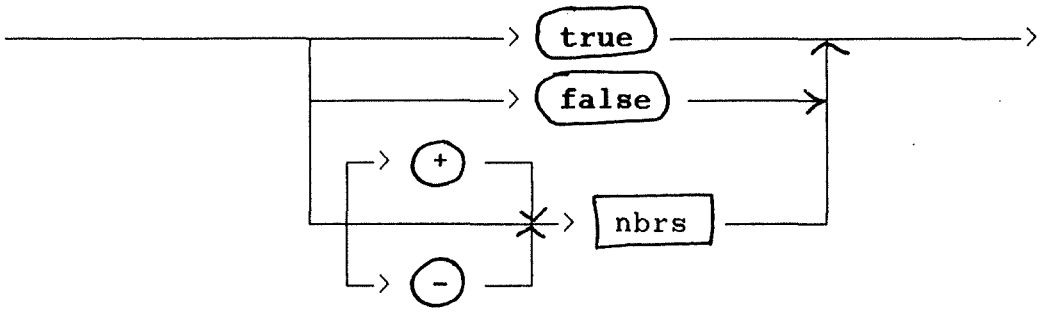
facteur



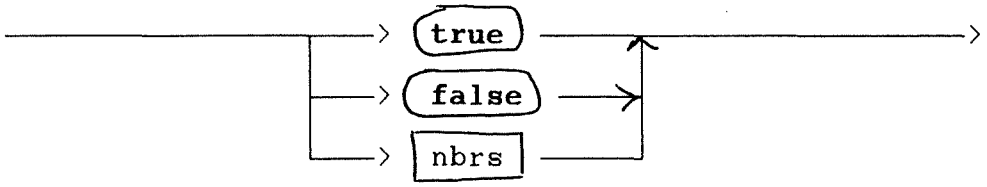
id



cst



cstss



nbrs